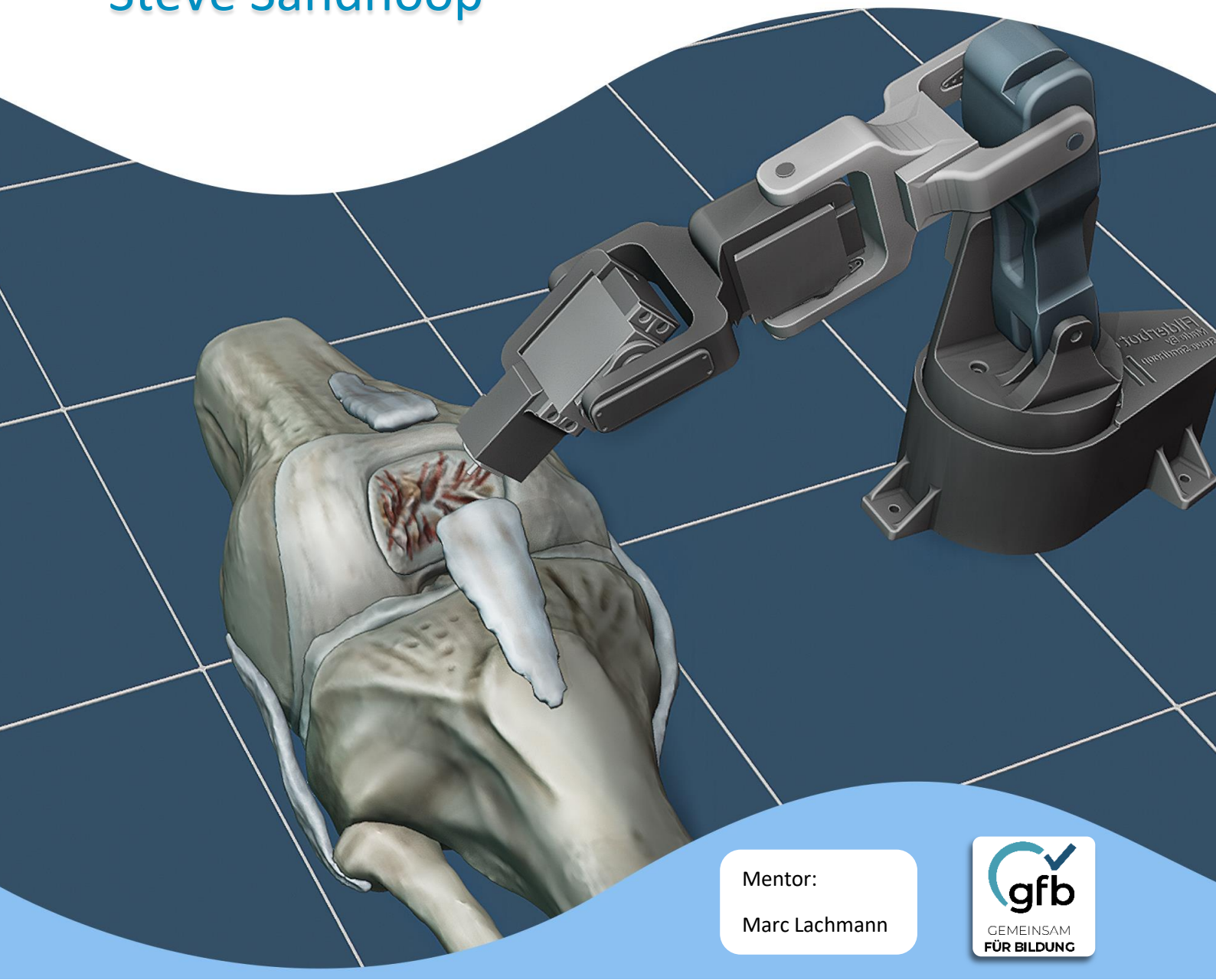
Entwicklung und Konstruktion eines chirurgischen Roboterarms zur Simulation einer roboterassistierten Patellofemoral-Gelenkersatz-Operation mittels eines virtuellen Zwillings als Kontrolleinheit.

Steve Sandhoop



Mentor:

Marc Lachmann

gfb
GEMEINSAM
FÜR BILDUNG

Inhaltsverzeichnis

Projektbeschreibung:

Die roboterassistierte Chirurgie ist eine aufstrebende Technologie, die in der modernen medizinischen Praxis immer häufiger eingesetzt wird. Durch den Einsatz von Robotern können komplexe Operationen präziser und effizienter durchgeführt werden. Dabei assistiert der Roboterarm dem Arzt, und ermöglicht eine höhere Präzision und Kontrolle während der Operation. Insbesondere bei orthopädischen Operationen kann die roboterassistierte Chirurgie zu einer verbesserten Behandlung beitragen. Diese Technologie erfordert eine enge Zusammenarbeit zwischen Mechanik und Software, um eine präzise Befehlsübertragung an den Roboterarm zu gewährleisten.

Abbildung: 1

Ziel des Projekts wird die Simulation einer Patellofemoral-Gelenkersatz Operation sein. Dabei wird der geschädigte Knorpel im Kniegelenk im Patellofemoral entfernt und durch ein Metallimplantat ersetzt. Dazu wird ein Bohrer verwendet, der am Roboterarm befestigt ist. Mit ihm kann der Arzt die geschädigten Bereiche manuell entfernen. Der Roboterarm unterstützt den Arzt bei der Operation, indem er nur einen bestimmten Bereich des Knies auswählt. Verlässt er diesen Bereich, wird der Bohrer sofort gestoppt, wodurch Komplikationen vermieden und die Operation präziser durchgeführt werden kann. Zu diesem Zweck verfügt der Roboterarm jederzeit über ein virtuelles Bild des Knieknochens, auf dem der zu entfernende Bereich ausgewählt werden kann.


Abbildung: 2

Die beschriebene Operation wandle ich in meiner Operation etwas ab, so dass der Roboterarm bei der Operation nicht assistiert, sondern durch die Auswahl des zu entfernenden Bereichs die Operation selbstständig durchführt. Ich habe mir diese Änderung überlegt, weil ich damit die Funktionsweise des Roboterarms weiterentwickel und nicht nur die Operationsmethode des vorhandenen Roboterarms kopiere.

Bei der Findung und Umsetzung der Projektidee hilft mir Herr Lachmann, Lehrer am Gymnasium Filder benden, der dort für die Informatik zuständig ist. Er ist mein Mentor für dieses Projekt und unterstützt mich beim 3D-Drucken der Modelle und bei der Suche nach Sponsoren, die mein Projekt finanziell unterstützen. Für mein Projekt hat mich Chirurg Dr Bertrams inhaltlich unterstützt. Finanziell unterstützen mich der Förderverein des Gymnasiums Filder Benden und die Katholische Karl-Leisner-Trägergesellschaft.

Abbildung: 3

Bei der Projektentwicklung handelt es sich um eine agile Projektentwicklung. Das bedeutet, dass sich die endgültige Idee im Laufe der Projektentwicklung und Umsetzung herauskristallisiert. Ziel dieser Art der Projektentwicklung ist die Erreichung kurzfristiger Ziele und nicht die Verfolgung eines langfristigen Plans. Dies ermöglicht eine größere Flexibilität bei der Umsetzung. Das Endprodukt verändert sich dynamisch während der Entwicklung, um den neuen angepassten Plänen, die während des Arbeitsprozesses gemacht wurden, zu entsprechen. Beispielsweise wurde in diesem Projekt die Benutzerschnittstelle während der Konstruktion des

Roboterarms dynamisch entwickelt, um sie an die neuen Anforderungen der neuen Version anzupassen. (Q1)

Um den beschriebenen Vorgang simulieren zu können, muss ich verschiedene Komponenten bauen und programmieren. Zuerst möchte ich meinen eigenen Roboterarm bauen, den ich selbst entwerfen und mit einem 3D-Drucker ausdrucken möchte. Er muss sich präzise bewegen können und am Ende einen Bohrkopf haben. Eine weitere Komponente ist die Kommunikation mit dem Roboterarm. Dazu müssen die Motoren des Roboterarms von der Software gesteuert werden können. Dafür verwende ich den Arduino Uno Chip, mit dem man sehr einfach jede Art von Motoren steuern kann. Damit kommuniziere ich mit der Unity Engine auf dem Computer. Die andere Komponente ist die Benutzerschnittstelle, die in Unity implementiert wird. Außerdem möchte ich ein eigenes Kniemodell erstellen, an dem die Operation durchgeführt wird.

Grundlagen vom Arduino:

Bevor wir uns mit dem spezifischen Arduino-Projekt beschäftigen, ist es wichtig, die grundlegenden Funktionen und Eigenschaften des Arduino-Chips zu verstehen, um einen Einblick in die Funktionsweise des Projekts zu erhalten.

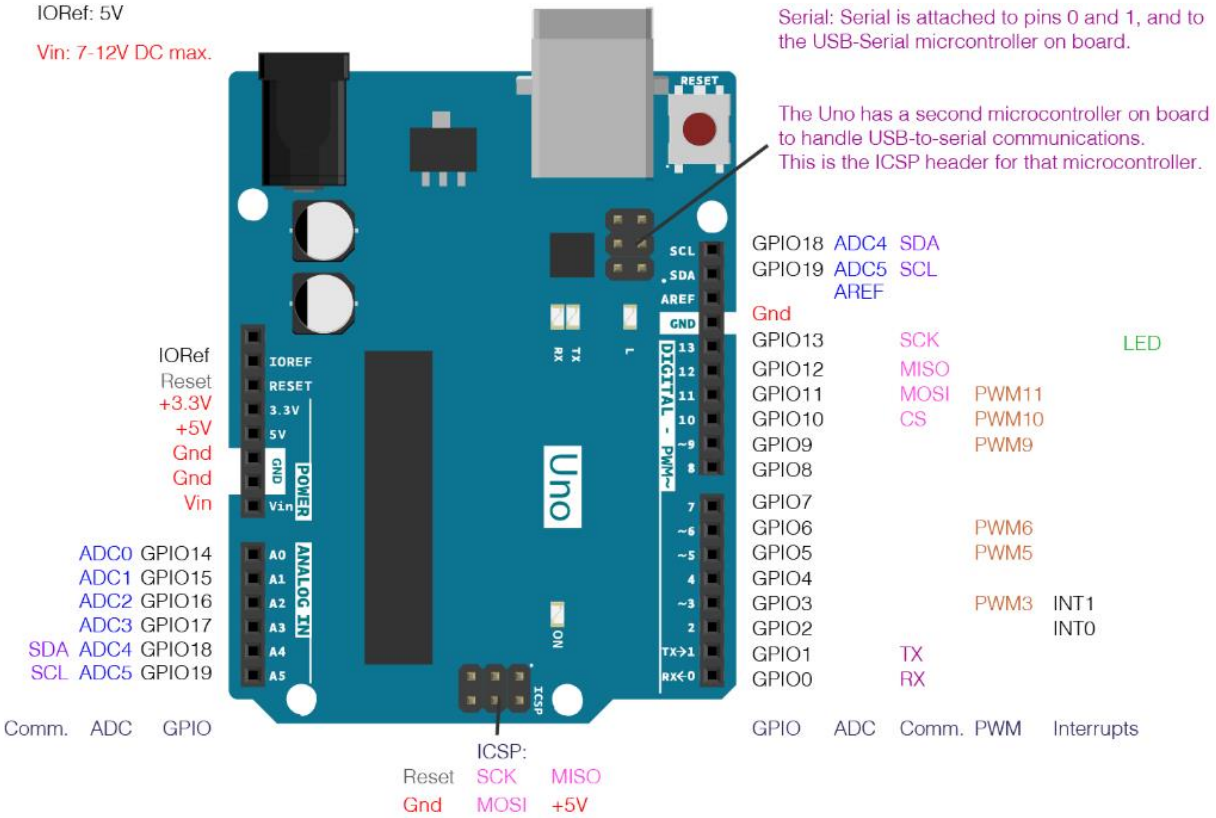


Abbildung: 4

Bei dem Arduino Uno Chip handelt es sich um einen programmierbaren Mikroprozessor, mit dessen Hilfe einfache Schaltungen und Motoren programmiert werden können.

Die rechten Pins sind programmierbare digitale Pins. Sie können jeweils als Eingang definiert werden, so dass beim Messen einer Spannung diese in einem Programm abgefragt werden kann, oder als Ausgang, um eine Spannung von 5V auszugeben. Die Pins in der linken unteren Ecke A0-A5 sind analoge Pins, mit denen einfache Eingänge abgefragt werden können. Die Pins in der oberen linken Ecke sind hauptsächlich für die Stromversorgung des Arduinos zuständig. Die Gnd (Ground) Pins sind Masseleitungen (Minuspol), die zusammen mit dem +5v oder +3.3v Pin (Pluspol) die entsprechende Spannung an andere Geräte übertragen können. Wenn der Strom von den digitalen Pins kommt, müssen die Gnd Pins entsprechend dem Strom geschlossen werden. (Q2, Q3)

Die Pins 9-13 haben noch die Möglichkeit, ein PWM-Signal auszugeben. PWM steht für Pulsweitenmodulation. Dieses Signal gibt Impulse aus. Die Grafik zeigt das Signal und seine Charakteristik.

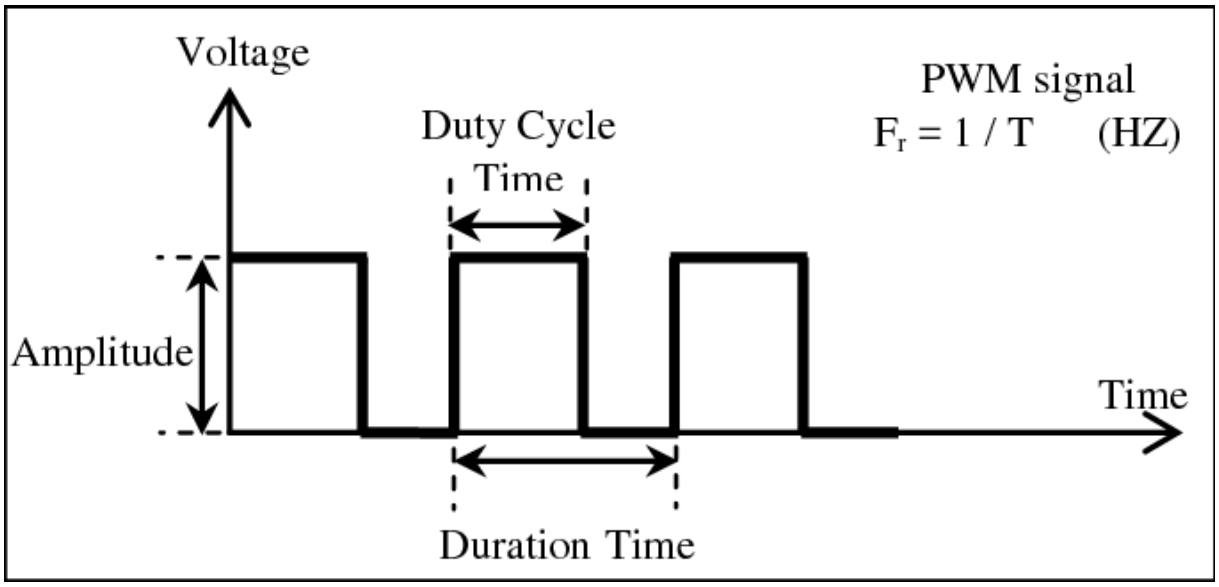


Abbildung: 5

Ein PWM-Signal (Pulsweitenmodulation) ist ein periodisches Signal, bei dem die Dauer des Signalimpulses (auch Impulsdauer oder Impulsbreite genannt) variiert wird, während die Periodendauer des Signals konstant bleibt. Das Signal wird üblicherweise als Rechteckwelle dargestellt, wobei die Dauer des Signalimpulses (in Prozent der Periodendauer) die Stärke oder Intensität des Signals darstellt. (Q4)

Die Pulsweitenmodulation wird im Allgemeinen zur Steuerung der Leistung oder Geschwindigkeit elektrischer Geräte verwendet, z. B. zur Steuerung der Drehzahl von Gleichstrommotoren oder der Helligkeit von LEDs. Die Änderung der Impulsbreite wird durch eine Steuerung verursacht, die das Signal ein- und ausschaltet, um den Mittelwert der Ausgangsleistung zu ändern. Durch die Änderung der Impulsdauer wird der Effektivwert der Spannung (oder des Stroms) geändert, was zu einer Änderung der Leistung oder Geschwindigkeit des angeschlossenen Geräts führt. (Q4)

Ein typisches Beispiel für die Anwendung der PWM ist die Drehzahlregelung von Gleichstrommotoren. Durch Anlegen eines PWM-Signals an den Eingang des Motors kann die Drehzahl des Motors gesteuert werden, indem die Impulsdauer des Signals variiert wird. Ein längerer Signalimpuls führt zu einem höheren Spannungsmittelwert, wodurch der Motor schneller läuft. Eine kürzere Impulsdauer führt zu einem niedrigeren Spannungsmittelwert, wodurch der Motor langsamer läuft.

Nun kommen wir zum Aufbau der Integrierten Entwicklungsumgebung kurz IDE (Integrated Developer Environment). In der IDE kann man zwischen den Ports der Arduinos umschalten, wenn man mehrere Arduinos gleichzeitig angeschlossen hat. Man kann Code kompilieren und mit dem Pfeil, der über ein USB-Kabel mit dem Computer verbunden ist, auf den Arduino laden. Im Texteditor programmiert man mit der Arduino eigenen Programmiersprache, die C ähnlich ist. Man verwendet immer die beiden Funktionen setup() und loop(), um ein Programm zu schreiben. In setup() schreibt man Codesegmente, die beim Start des Arduinos ausgeführt werden sollen. Zum Beispiel kann man mit pinMode die digitalen Pins konfigurieren, indem man die Pins 0-13 und den Typ angibt. In loop wird der Code während des Betriebs des Arduinos immer wieder ausgeführt. Hier kann man, wie im Beispiel zu sehen, die konfigurierten Pinsignale mit HIGH = 5V und LOW = 0V senden. Um eine Pause zwischen den Schritten zu haben, kann man mit Delay in Millisekunden den Ablauf für diese Zeit pausieren. Das Beispiel lässt eine Lampe einmal pro Sekunde blinken. Der Schaltplan sieht wie folgt aus. (Q5)

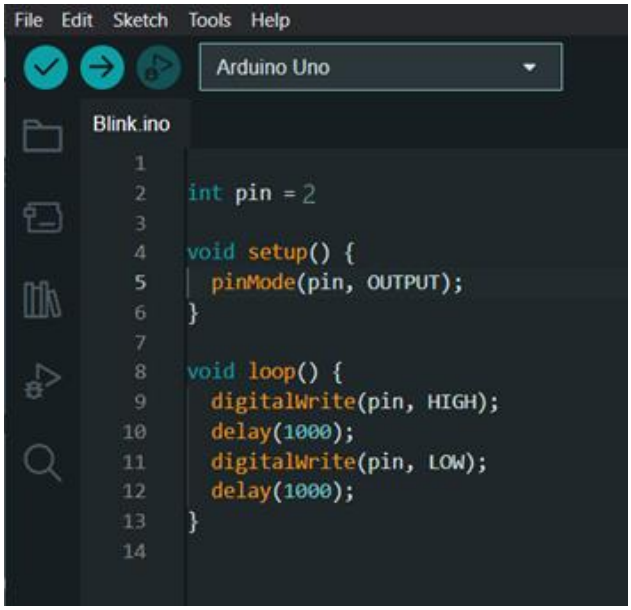


Abbildung: 6

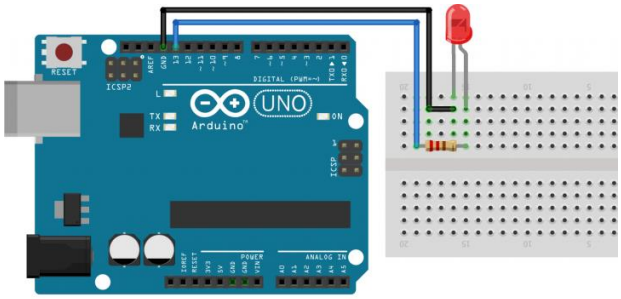


Abbildung: 7

Mit den PWM-Pins können Servomotoren angesteuert werden. Mit der Servobibliothek kann man die Servomotoren einfacher ansteuern, da man das PWM-Signal nicht selbst erzeugen muss. Dazu importiert man die Servobibliothek mit #include <Servo.h>. Anschließend kann man Servo Objekte erzeugen, diese in Setup() einem Pin zuweisen und in loop() die Gradzahl angeben, um die sich der Servo drehen soll. So weiß der Motor immer, wie er gedreht wird und dreht sich immer in die gleiche Richtung. Der Nachteil des Servomotors ist, dass er zwar einfach zu bedienen ist, sich aber nur um 180 Grad drehen kann. (Q6 Q7)

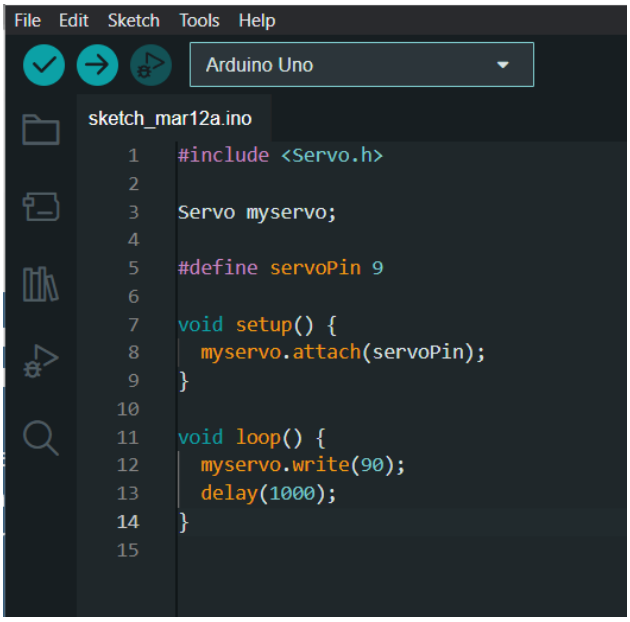


Abbildung: 8

Datenübermittlung von Unity zum Arduino:

Jetzt kann ich die Bewegung der Servomotoren mit dem Arduino programmieren. Die Schaltskizze ist in Abbildung 9 zu sehen. Aber ich kann nur eine Bewegungssequenz schreiben und sie noch nicht interaktiv steuern. Man könnte ein Potentiometer anschließen, um die Motoren zu drehen. Ich möchte sie aber mit meinem Laptop per Software steuern können. Um Signale im Arduino zu empfangen, muss man eine Serial Baud Rate einstellen. Dazu schreibt man in Setup() Serial.begin(9600). Das ist praktisch der Kanal, über den man mit dem Computer kommuniziert.

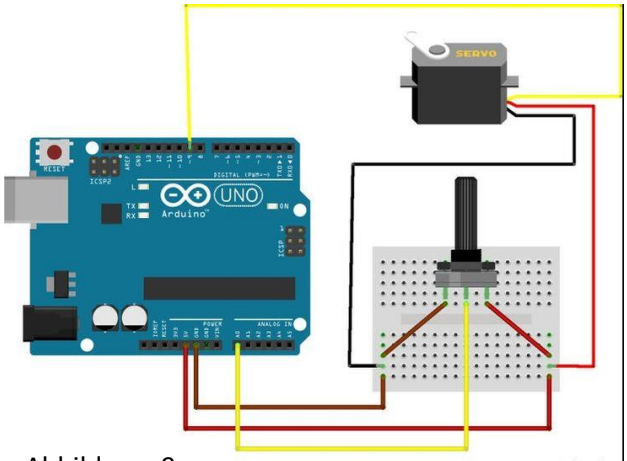


Abbildung: 9

Für das Lesen der Signale stehen verschiedene Methoden zur Verfügung:

Mit “String input = Serial.readString();“ kann man eine Folge von Buchstaben und Zahlen empfangen und diese verwenden, um den Servomotoren mitzuteilen, wie sie sich drehen sollen. Das erste Skript ist in Abbildung 10 zu sehen. Damit kann der Arduino Signale empfangen und die Motoren entsprechend drehen. Ich habe noch ein Potentiometer angeschlossen, das ich im Code so verändern kann, dass ich damit den Servomotor testweise noch steuern kann. Um nun die Signale zu senden, kommt Unity zum Einsatz. Unity (Q8) ist eine 3D/2D-Engine, mit der man virtuelle Umgebungen erstellen kann, z.B. Videospiele. Ich möchte den Roboter später mit einem digitalen Zwilling steuern können, dafür ist Unity perfekt geeignet. Unity selbst kann nicht mit dem Arduino kommunizieren. Dafür braucht man ein zusätzliches Add-on, das als API zwischen dem Arduino und Unity fungiert. Für mein Projekt verwende ich das kostenlose Open Source Add-on Ardity (Q9).

Mit Ardity kommen zwei Skripte einmal der Serial Controller, der für die Verbindung

```
#include <Servo.h>

Servo motor1;

int val = 0;
int stepSize = 36;
int analogPin = A0;

bool analog = true;

void setup(){
  motor1.attach(9);
  Serial.begin(9600);
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop(){

  if (analog){
    val = analogRead(analogPin);
    val = map(val, 0, 1023, 0, 180);
    motor1.write(val);
  }
  else if (Serial.available()){

    int input = Serial.readString().toInt();
    Serial.println(input);

    val = input;

    motor1.write(val);
  }

  delay(100);
}
```

Abbildung: 10

zuständig ist. Hier muss man die gleiche Serial Baud Rate einstellen, die ich vorher im Arduino eingestellt habe.

```
2 Verweise
void checkArduinoConnection(SerialController inputArduino, string dataType)
{
    string message = inputArduino.ReadSerialMessage();

    if (message == null)
        return;

    // Check if the message is plain data or a connect/disconnect event.
    if (ReferenceEquals(message, SerialController.SERIAL_DEVICE_CONNECTED))
        Debug.Log("Connection established "+ dataType);
    else if (ReferenceEquals(message, SerialController.SERIAL_DEVICE_DISCONNECTED))
        Debug.Log("Connection attempt failed or disconnection detected "+ dataType);
    else
        Debug.Log("Message arrived: " + dataType + " "+ message);
}

1 Verweis
public void sendSignal(string eingabe)
{
    if (DebugMode)
    {
        Debug.Log("Sending:  " + eingabe);
    }
    serialController_AxialData.SendSerialMessage(eingabe);
}
```

Abbildung: 11

Das zweite Skript ist der Arduino Transmitter, zu sehen in Abbildung 11, der mit Funktionen kommt, mit denen man Signale senden und empfangen kann. Die Funktion checkArduinoConnection() testet die Verbindung und gibt Informationen über den Vebindungsstatus wieder. Mit der Funktion sendSignal() lassen sich Nachrichten zum Arduino senden.

Mit diesen Skripten ist die Grundlage für die Datenübertragung geschaffen. Um einen kleinen Roboter zu haben, an dem ich die Software testen kann, habe ich einen kleinen Roboterarm mit 4 Tower Pro MG90S Micro Servo Motoren und Dateien für den Lasercutter zum Ausschneiden der Teile gekauft.

Zur Steuerung habe ich eine rudimentäre Benutzeroberfläche in Unity erstellt (Abbildung 13).

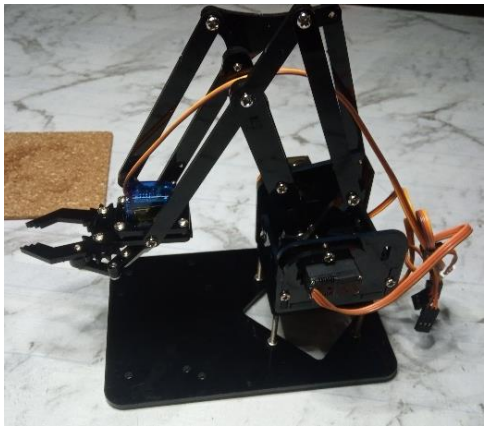


Abbildung: 12

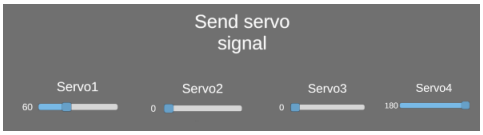


Abbildung: 13

Bei diesem ersten Test fiel mir auf, dass er relativ spät auf das Signal reagierte, obwohl ich eine relativ hohe Taktrate für die Kommunikation zwischen Arduino und Unity eingestellt hatte. Schließlich fand ich heraus, dass der Fehler in der Verwendung der Funktion Serial.readString(); lag. Anscheinend arbeitet diese Funktion nicht asynchron, d.h. sie wartet immer, bis das Signal angekommen ist, bevor sie ein neues sendet. Das ist ungünstig, wenn ich 4 Servos gleichzeitig ansteuern will, da es auf die anderen wartet. Um eine asynchrone Datenübertragung zu erreichen, muss ich mit Serial.read(); das Signal als reine Bytefolge abfangen. Zuerst muss ich die Bytes in einen String umwandeln. Dazu habe ich zwei Funktionen benutzt. Die erste Funktion recWithEndMarker() liest die Bytes und fügt sie in eine statische Bytevariable ein, bis der Endmarker erreicht ist. Dies ist einfach die Änderung der Signalzeile. Am Ende der Funktion wird die Signalzeile im Array receivedChars gespeichert. Dieses kann ich dann in einer anderen Funktion mit einer For-Schleife durchlaufen und habe einen String aus dem Signal. Das Signal ist so aufgebaut, dass ich zuerst einen Buchstaben sende, der angibt, welchen Servomotor ich steuern möchte. Dann folgt eine Zahlenfolge, um die Gradzahl einzustellen. Beispiel für ein Signal: „a140“ - dreht den ersten Motor um 140°.

```
void showNewNumber() {
    if (newData == true) {

        Serial.println(receivedChars);


        String temp = "";

        for(int j = 1; j < strlen(receivedChars); j++){
            temp += receivedChars[j];
        }
```

Abbildung: 14

```
void recvWithEndMarker() {
    static byte ndx = 0;
    char endMarker = '\n';
    char rc;

    if (Serial.available() > 0) {
        rc = Serial.read();

        if (rc != endMarker) {
            receivedChars[ndx] = rc;
            ndx++;
            if (ndx >= numChars) {
                ndx = numChars - 1;
            }
        }
        else {
            receivedChars[ndx] = '\0'; // terminate the string
            ndx = 0;
            newData = true;
        }
    }
}
```

Abbildung: 15

Um später mehrere und stärkere Servos ansteuern zu können, benötige ich ein Shield für den Arduino. Ein Shield für den Arduino ist eine Platine, die auf den Arduino-Controller gesteckt wird oder verbunden wird, um dessen Funktionen zu erweitern oder zu ergänzen. Ein Shield bietet eine einfache Möglichkeit, um zusätzliche Elektronik-Komponenten hinzuzufügen, ohne dass man aufwendige Schaltkreise auf einem Breadboard oder einer Lochrasterplatine entwerfen und verdrahten muss.

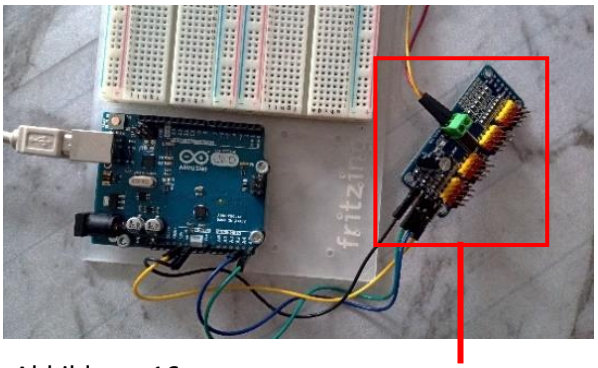


Abbildung: 16

PCA-Chip

Der Arduino verträgt keine höhere Strombelastung, die mehrere Servos benötigen würden, weswegen das Shield über Spannungsregler verfügt. Das Shield nimmt mit einer zusätzlichen Batterie oder einem Netzteil zusätzlichen Strom auf, um die benötigten Stromstärken zu liefern. Das hier verwendete Shield ist der Adafruit PCA9685 16-Channel Servo Driver (Q10.2), wie in Abbildung 16 rechts zu sehen. Der Adafruit PCA9685 16-Channel Servo Driver ist ein I²C-basierter PWM-Controller, der für die Steuerung von bis zu 16 Servos oder anderen elektromechanischen Geräten verwendet werden kann. Der Chip ermöglicht die unabhängige Steuerung von 16 PWM-Kanälen, die für die Positionierung von Servomotoren, die Regelung von Helligkeit bei LED-Streifen oder für andere ähnliche Anwendungen verwendet werden können. Die PWM-Frequenz kann bis zu 1,6 kHz eingestellt werden, was für viele Anwendungen ausreichend ist.

Der Adafruit PCA9685 ist für den Einsatz mit dem Arduino oder anderen Mikrocontrollern konzipiert und kann über den I²C-Bus seriellen Datenbus kommunizieren. Die Steuerung erfolgt über eine einfache Bibliothek, die vom Benutzer in der Arduino-IDE eingebunden wird. Die Bibliothek bietet Funktionen für die Einstellung von Servo-Positionen oder für die Steuerung von PWM-Ausgängen.

Der PCA-Chip kann mit der Adafruit_PWMservoDriver Bibliothek gesteuert werden. Mit einem Adafruit_PWMServoDriver Objekt können alle angeschlossenen Servos einfach mit der Funktion setPWM(Servo Input, Tick Rate, PWM Signal) gesteuert werden. Zuerst gibt man die Nummer des Servos ein und am Ende die Position, an der es gedreht werden soll. Dabei wird das PWM-Signal verwendet, also statt 0°-360° die Werte 150-600. 150-600 sind die verwendeten PWM Frequenzen für den Chip. Das muss ich bei der Signalübertragung berücksichtigen und die Gradzahlen in PWM-Werte umwandeln. (Q10.1, Q10.2)

```
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>

Adafruit_PWMServoDriver pcaChip = Adafruit_PWMServoDriver();

#define servoMIN 150
#define servoMAX 600

void setup() {
  Serial.begin(9600);
  pcaChip.begin();
  pcaChip.setPWMFreq(60);

  pcaChip.setPWM(0, 0, 400);
  pcaChip.setPWM(1, 0, 400);
  pcaChip.setPWM(2, 0, 400);
}

void loop() {
  pcaChip.setPWM(0, 0, 500);
  pcaChip.setPWM(1, 0, 500);
  pcaChip.setPWM(2, 0, 500);

  delay(2000);

  pcaChip.setPWM(0, 0, 400);
  pcaChip.setPWM(1, 0, 400);
  pcaChip.setPWM(2, 0, 400);

  delay(2000);
}
```

Abbildung: 17

Digitaler Zwilling:

Die erste Steuerung für den Roboterarm ist fertig. Ich möchte ihn aber nicht nur mit Reglern bewegen, sondern eine komplette digitale Version des Roboters haben, die ich bewegen kann. Dazu benutze ich Blender (Q11), ein kostenloses 3D-Modellierungsprogramm, mit dem ich den Roboter nachbaue. Dieses Modell kann ich dann mit einem Rig versehen. In der Computergrafik bezieht sich ein Rig auf eine Sammlung von virtuellen Steuerelementen oder Knochen, die zur Steuerung der Bewegung und Deformation eines 3D-Modells verwendet werden.

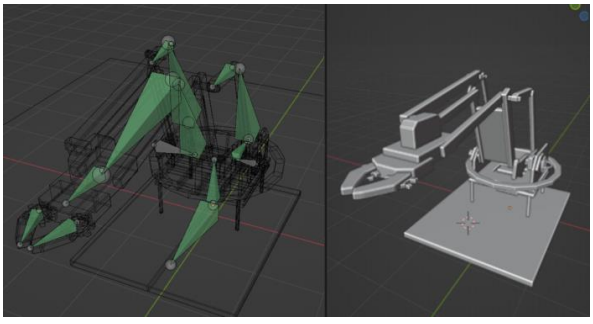


Abbildung: 18

Ein Rig wird typischerweise erstellt, um einem Modell Beweglichkeit und Realismus zu verleihen. Ein Rig kann als eine Art virtuelles Skelett betrachtet werden, das mit dem Modell verbunden ist und es ermöglicht, die Position, Rotation und Skalierung von verschiedenen Teilen des Modells präzise zu steuern. Dies kann nützlich sein, um beispielsweise die Bewegungen eines animierten Charakters zu steuern oder eine komplexe Maschine in einer 3D-Simulation zu bewegen. (Q12)

Das erstellte Modell kann ich dann in Unity einfügen. Bewegen kann ich den Rig nur in Blender, da in Unity den Rig nur als Basis für die Bewegung dient. Da ich meine eigene Benutzeroberfläche in Unity programmieren möchte, muss ich zuerst Controller für das Rig programmieren.

Die Bedienelemente, wie in Abbildung 19 zu sehen, sind als Ringe dargestellt, die angeklickt werden können, damit man sie um eine bestimmte Achse drehen kann. Ich habe Ringe als Visualisierung gewählt damit deutlich wird, dass die Kontroller als Drehachsen verwendet werden.

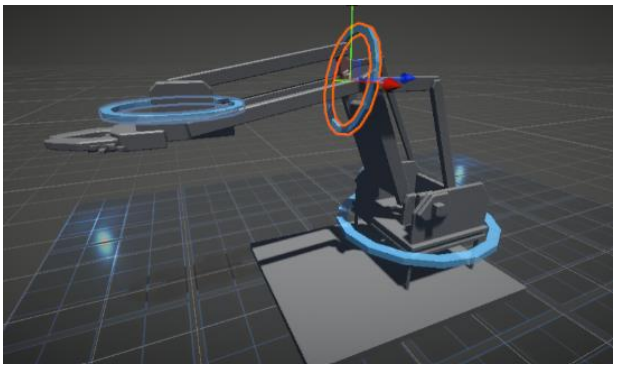


Abbildung: 19

Um die Controller auszuwählen, benutze ich einen RayCast. Ein RayCast ist eine Funktion, die einen Strahl aussendet und das getroffene Objekt zurückgibt. Dieser Strahl wird ständig von der Maus gesendet und kann so überprüfen, ob sich die Maus über einen Controller bewegt und dieser dann angeklickt wird.

```
Ray ray = Camera.main.ScreenPointToRay(Inputs.mousePosition);
RaycastHit hit;
if (Physics.Raycast(ray, out hit, 1000, MainRayCastMask))
{
```

Abbildung: 20

```
if (selected)
{
    float angle = 0;
    Vector3 temp = Vector3.zero;
    angle = inverter*Mathf.Clamp(selectMouseAxis - dragMouse, -100, 100) * manage.movementSpeed * Time.deltaTime;

    if(rotateAxis == "x")
    {
        temp = new Vector3(angle, 0, 0);
    }
    else if (rotateAxis == "y")
    {
        temp = new Vector3(0, angle, 0);
    }
    else if (rotateAxis == "z")
    {
        temp = new Vector3(0, 0, angle);
    }

    transform.localEulerAngles += temp;
}
```

Abbildung: 21

Dann wird der entsprechende Controller ausgewählt und mit der relativen Mausbewegung auf der entsprechenden Achse, die ich für jeden Controller auswähle, gedreht.

```
Vector3 min = new Vector3(transform.localEulerAngles.x, transform.localEulerAngles.y, minAngle);
Vector3 max = new Vector3(transform.localEulerAngles.x, transform.localEulerAngles.y, maxAngle);
ClampAngle(transform.localEulerAngles.z, min, max);
```

Abbildung: 22

Damit sich die Regler nicht über 360° oder 0° hinaus drehen können, begrenze ich sie mit der Funktion Mathf.Clamp() auf diese Werte. Nun kann das Modell mit den Controllern bewegt werden. Die Controller sollen nun durch die Winkel der Achsen ersetzt werden, so dass der physikalische Roboterarm den virtuellen nachahmt. Dazu verknüpfe ich zunächst alle Achsen mit einem ServoAssembly()-Skript, das ein Array der Klasse Servo() speichert. Die Klasse Servo() speichert Werte über eine Achse die rechts zu sehen sind. Diese sind dafür da um die Achsen einzeln einstellen zu können um z.B. die Achsen Richtung x,y,z oder die Begrenzungen einzustellen. Am wichtigsten ist natürlich die Variable servo, welche das Transform Objekt der

```
1 Verweis
public class Servo
{
    public Transform servo;
    public string axis = "x";

    public float angleOffset;
    public bool invertAngle;
    public float clampMin = 0;
    public float clampMax = 180;
}
```

Abbildung: 23

```
float convertToSignal(int Servo, string axis)
{
    //calibration
    float angleDirection = 1;
    if (servoAssembly.Servos[Servo].invertAngle)
    {
        angleDirection = -1;
    }

    float servoValue = 0;

    if (axis == "x")
    {
        servoValue = angleDirection * (servoAssembly.Servos[Servo].servo.localEulerAngles.x + servoAssembly.Servos[Servo].angleOffset);
    }
    else if (axis == "y")
    {
        servoValue = angleDirection * (servoAssembly.Servos[Servo].servo.localEulerAngles.y + servoAssembly.Servos[Servo].angleOffset);
    }
    else if (axis == "z")
    {
        servoValue = angleDirection * (servoAssembly.Servos[Servo].servo.localEulerAngles.z + servoAssembly.Servos[Servo].angleOffset);
    }

    float servoValueClamped = Mathf.Clamp(servoValue, servoAssembly.Servos[Servo].clampMin, servoAssembly.Servos[Servo].clampMax);

    return servoValueClamped;
```

Abbildung: 24

Achse speichert, da hier die Rotation abgefragt werden kann. Diese Werte benutze ich um das Signal für die eingestellte Achse einzustellen.

Zuerst wird abgefragt, ob die Achse invertiert dargestellt werden soll. Dann wird die eingestellte Richtung x,y oder z abgefragt. Dann kann mit servoAssembly[i].Servos.servo.rotation.localEulerAngles die entsprechende Richtung in servoValue zwischengespeichert werden. Dazu wird jeweils ein AngleOffset für weitere Flexibilität gesetzt, falls die Winkel zwischen Digital und Physical nicht ganz übereinstimmen, z.B. dass beide sich richtig drehen, nur der virtuelle um 90° vorauseilt. Am Ende gibt die Funktion den begrenzten Winkel der Achse zurück.

Diese Funktion kann dann verwendet werden, um die Winkel aller Achsen in einem float Array „values“ zu speichern. Dieses Array enthält dann alle Werte der Achsen in Echtzeit. Diese kann ich mit dem Arduino Transmitter an den Arduino senden. Um die Werte auch im Arduino den jeweiligen Achsen zuzuordnen, durchlaufe ich das Array values mit einer for Schleife und gebe mit einer char Variablen „c“ dem Signal den Buchstaben, der für die Achse steht. Das Signal soll nur gesendet werden, wenn sich der Wert geändert hat. Dazu speichere ich nach dem Senden eines Signals den Wert von values in prevalues und prüfe beim erneuten Senden, ob diese gleich sind. Dadurch wird verhindert, dass der Roboter den Signalen hinterherhinkt.

```
char c = 'a';

for (int i = 0; i < servoAssembly.Servos.Length; i++)
{
    if (values[i] != prevalues[i])
    {
        sender.sendSignal(c + values[i].ToString());
    }
    prevalues[i] = values[i];
    c++;
}
```

Abbildung: 25

Damit kann ich den Roboter bewegen und habe einen digitalen Zwilling, dessen Bewegungen synchron sind.

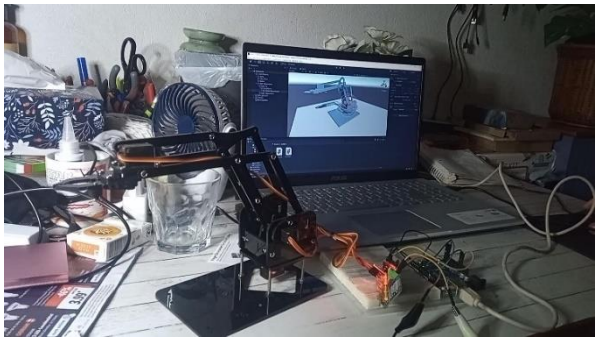


Abbildung: 26

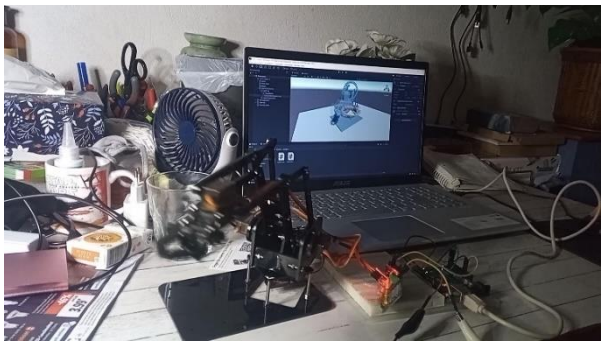


Abbildung: 27

Konstruktion des Roboterarms

Das wichtigste Element der Simulation ist der Roboterarm. Als vorläufiges Modell hatte ich den kleinen Bausatz eines Roboterarms gekauft. Dieser war nur für Testzwecke zu gebrauchen, da er einen sehr großen Bewegungsspielraum hatte, was zu einem Wackeln und einer ungenauen Bewegung der Achsen führte. Außerdem war er natürlich viel zu klein für die geplante Operation. Ein neuer, präziserer und größerer Roboterarm musste her.

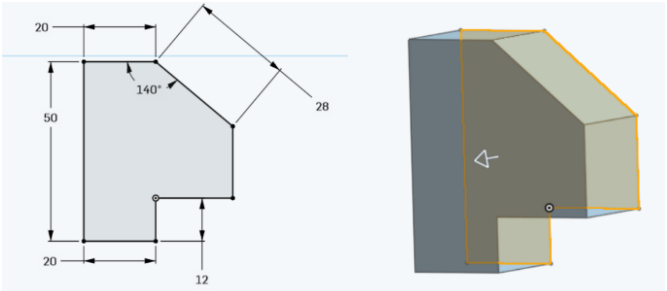


Abbildung: 28

Als Konstruktionssoftware benutzt man CAD (Computer Aided Design) Programme, die für die Konstruktion mit genauen Maßen und Konstruktionsplänen geeignet sind. Für meine Konstruktion verwende ich die Online-Software Onshape von PTC (Q13). Onshape funktioniert als SaaS (Software as a Service), das heißt, es läuft komplett online und muss nicht auf dem Computer installiert werden. Mit Onshape wird eine Skizze auf den Oberflächen von Objekten oder auf den Ausgangsebenen Top, Front, Right erstellt. Diese Skizze kann mit Maßen und Begrenzungen angepasst werden. Um aus den Skizzen 3D-Modelle zu erstellen, können diese durch Hinzufügen oder Entfernen von Volumenkörpern in den 3D-Raum erweitert werden.

Um die Motoren für meinen Roboterarm auswählen zu können, muss man zunächst wissen, welche Arten von Motoren es gibt. Grundsätzlich gibt es 3 verschiedene Kategorien von Motoren, die für unterschiedliche Funktionen eingesetzt werden können: Gleichstrommotoren, Servomotoren und Schrittmotoren. Der Gleichstrommotor ist der einfachste der drei. DC steht für Direct Current und bedeutet, dass er Gleichstrom als Energiequelle verwendet. Der Aufbau ist relativ einfach, der Rotor baut durch Spulen auf beiden Seiten ein Magnetfeld auf und der Stator baut durch Spulen, die sich abstoßen, ebenfalls ein Magnetfeld auf. Durch die Abstoßung dreht sich der Motor und der Kommutator bewirkt, dass sich das Magnetfeld nach einer Umdrehung umkehrt. Dies führt zu einer kontinuierlichen Drehung des Rotors. Der Gleichstrommotor ist ein einfacher Elektromotor, der sich nur in eine Richtung dreht. Durch Umpolen ändert sich die Drehrichtung. Um die Polarität zu steuern, kann eine H-Brücke verwendet werden, die die Polarität durch Signale ändert. (Die Funktionalität einer H-Brücke wird im genauen auf Seite 21 erklärt). Aufgrund seiner Einfachheit eignet sich der Gleichstrommotor am besten für Funktionen wie Bohrmaschinen oder andere Maschinen, die sich kontinuierlich in eine Richtung drehen. (Q14)

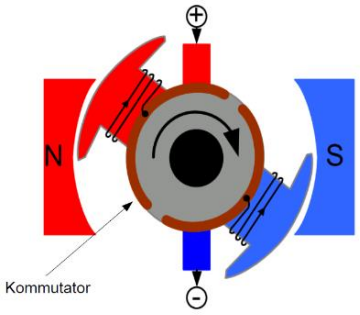


Abbildung: 29

Der Schrittmotor ist eine Erweiterung des Gleichstrommotors, der mehr Spulen im Stator hat. Dadurch kann er sich langsamer und genauer drehen als der Gleichstrommotor. Er dreht sich also in Schritten. Die Schrittweite in Grad gibt dann an, wie genau sich der Motor drehen kann. Da nun viele Spulen angesteuert werden müssen und zum Umpolen in die andere Richtung eine H-Brücke benötigt wird, muss der Schrittmotor mit einem Schrittmotortreiber angesteuert werden. An diesen können Signale gesendet werden, die den Schrittmotor in die entsprechende Richtung mit der entsprechenden Geschwindigkeit drehen. Aufgrund seiner präzisen Drehung wird er für 3D-Drucker und präzise arbeitende Roboterarme verwendet. Allerdings ist er relativ schwach, da wie beim Gleichstrommotor nur das Magnetfeld der Spulen für die Kraft des Motors verantwortlich ist. Aus diesem Grund sind für die Bewegung von Roboterarmen große Motoren oder ein ausgeklügeltes Getriebe erforderlich. (Q15)

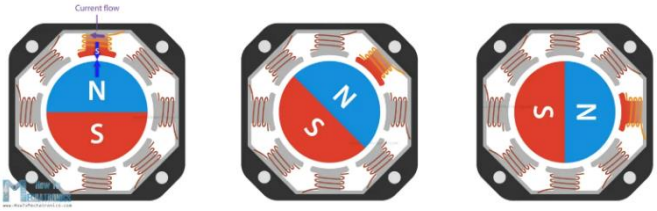


Abbildung: 30

Der Servomotor ist im Vergleich zu den anderen Motoren eine komplette Einheit. Er enthält einen Gleichstrommotor, der ein Zahnradgetriebe antreibt. Dadurch ist er stärker und kann mehr Last tragen. An den Zahnrädern ist außerdem ein Potentiometer angebracht, das misst, in welchem Winkel sich der Motor gerade befindet. Durch die Zahnräder hat der Motor mehr Kraft, ist aber in seiner Drehung begrenzt. Die meisten

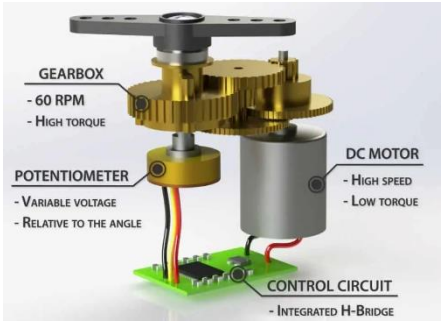


Abbildung: 31

Modelle können sich von 0-180° oder 0-360° drehen. Der Gleichstrommotor und das Potentiometer werden durch einen eingebauten Chip gesteuert, so dass kein zusätzlicher Treiber benötigt wird, um den Motor zu steuern. Wie der Schrittmotor wird auch der Servomotor hauptsächlich in der Robotik eingesetzt. Bei ihm liegt der Schwerpunkt mehr auf Kraft als auf Präzision, was bei der Funktion des Roboterarms berücksichtigt werden muss. (Q16)

Für meinen Roboterarm habe ich zuerst Servomotoren verwendet, da ich sie zuerst kennengelernt habe und Sie sind in vielen der Robotik-Projekte, die ich gesehen habe, zum Einsatz gekommen. Außerdem sind sie aufgrund ihrer Bauweise einfach zu handhaben und der Motor weiß immer, in welchem Winkel er sich befindet, was bei den anderen Motoren nicht der Fall ist. Trotzdem musste ich später feststellen, dass die viel präziseren Schrittmotoren für mein Projekt besser geeignet sind.

Zuerst werde ich die Konstruktion des ersten Roboterarms erläutern, der mit Servomotoren realisiert wurde. Für die erste Version habe ich als Grundlage ein kostenloses Modell eines Roboterarms verwendet. Ich habe mich für das Modell EEZYbotARM MK2 (Q17) entschieden. Der Vorteil dieses Modells ist, dass es bereits als Onshape-Projekt vorliegt und ich keine einzelnen Step-Dateien einfügen muss. Das Design des Roboters verwendet drei MG995 Servomotoren und einen Tower Pro MG90S am Ende des Arms. Die MG995 Servomotoren sind viel stärker als die kleinen MG90S Motoren, deshalb können sie das größere Gewicht des Roboterarms tragen. Bei dieser Konstruktion befindet sich am Ende des Arms ein Greifer mit dem Servomotor. Das ist die erste Stelle, die ich überarbeitet habe, dass ich den Greifer durch einen Gleichstrommotor ersetzt habe und zusätzlich einen Servomotor für die vertikale Bewegung des Roboterarms.

Abbildung: 32

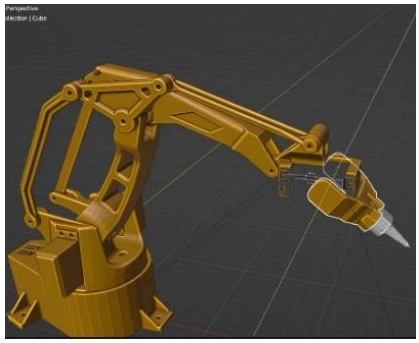


Abbildung: 33

Durch meine geplanten Modifikationen habe ich festgestellt, dass der ursprüngliche Entwurf nur dafür gedacht war einen kleinen MG90-Servomotor zu tragen. Dann habe ich eigentlich angefangen, das ganze Design zu überarbeiten, dass ich es am Ende selbst modelliert habe. Das einzige, was ich behalten habe, war die Basis, die mit Zahnrädern und einem Kugellager gebaut wurde. Dadurch habe ich mich mehr mit Kugellagern beschäftigt und wie sie funktionieren.

Im Prinzip gibt es einen äußeren und einen inneren Ring, die nicht miteinander verbunden sind oder sich berühren. Das Ziel eines Kugellagers ist es, die Bewegungen weicher zu machen und Reibung zu vermeiden. Deshalb sind die Verbindungen zwischen den Ringen mit glatten Stahlkugeln versehen, die durch ihre Rotation und ihre glatte Oberfläche Reibung vermeiden.

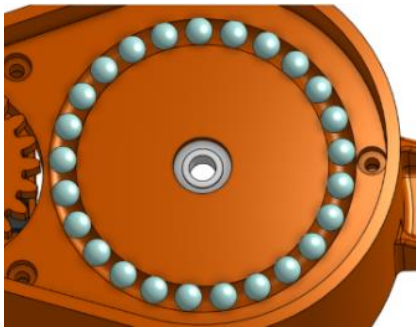


Abbildung: 34

Die Achsen des Roboterarms habe ich durch diese Klammern ersetzt, die ich konstruiert habe. In diese Klammer wird ein Servomotor eingesetzt, der durch seine Rotation das Teil, in dem er sich befindet, dreht. Dieses Teil kann beliebig oft vervielfältigt werden, um als Kette weitere Achsen des Roboterarms zu bilden.

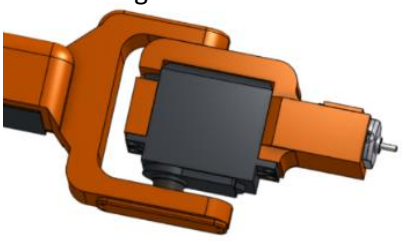


Abbildung: 35

Als ich die Achse konstruierte, hatte ich nicht daran gedacht, dass ich die Servomotoren so nicht hineinbekomme, und musste eine Lösung finden. Als Lösung habe ich mir überlegt, dass ich den Servo Tip, also diese Art von

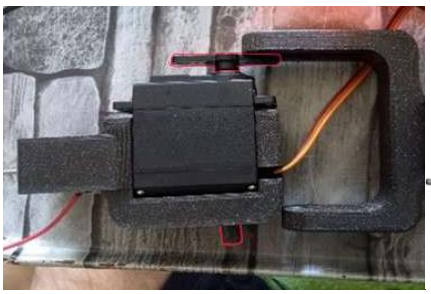


Abbildung: 36

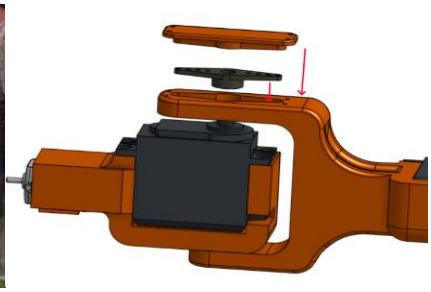


Abbildung: 37

Propeller, zuerst abnehmen kann, um den Servomotor in die Klammer zu bekommen. Das Problem ist dann, dass ich ihn nicht mehr auf das Servo setzen kann. Also mache ich oben in die Klammer ein Loch für den Servo Tip, durch das ich ihn wieder draufsetzen kann. Dazu noch eine Kappe und das Problem ist gelöst.

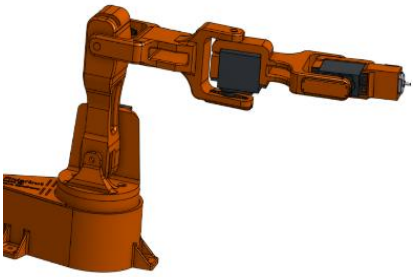


Abbildung: 38

Der Roboterarm, der zuerst gedruckt wurde, ist in Abbildung 36 zu sehen. Es ist ein 5DOF Roboterarm, was für Degree of Freedom steht, d.h. er kann sich in 5 Achsen bewegen. Als ich ihn das erste Mal bewegt habe, habe ich schnell gemerkt, dass durch die langen Achsen die Bewegungsfreiheit nicht optimal ist. Außerdem war er schon relativ schwer und kippte durch die langen Achsen etwas nach vorne. Deshalb habe ich dann 2 Teile gekürzt und ausgetauscht. Dadurch kam er besser auf den Boden und kippte nicht mehr.

Um ihn durch die Software zu bewegen, muss sich der Roboterarm einfach zu einem Punkt bewegen können, den man ihm vorgibt. Dazu benutzt man Inverse Kinematics, das ist ein Algorithmus, um die Winkel der Achsen eines Roboterarms zu erhalten.

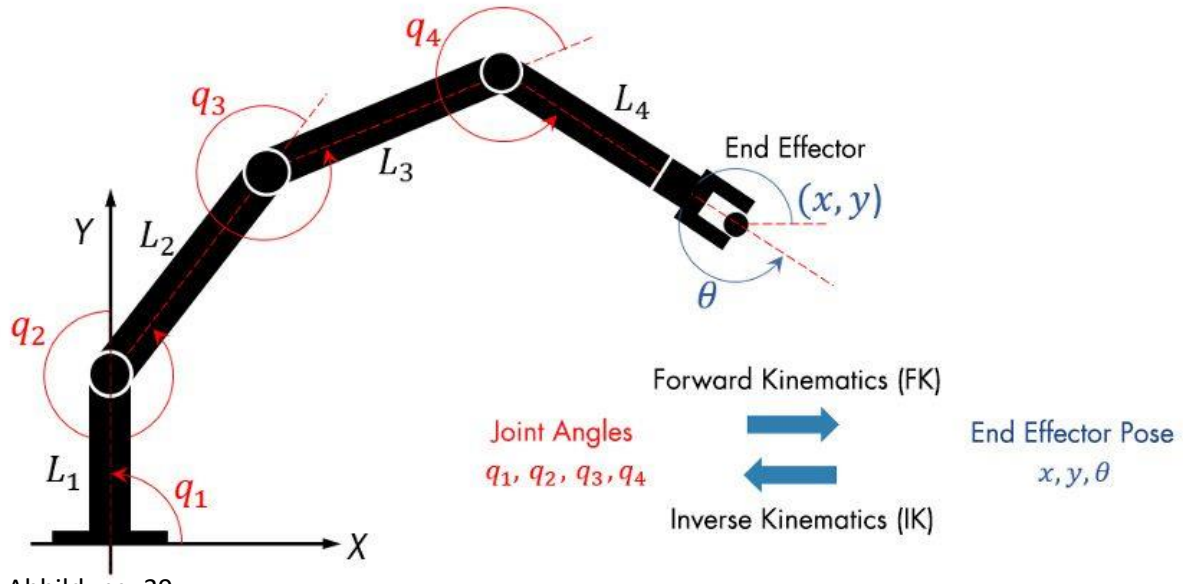


Abbildung: 39

Um Inverse Kinematics zu erklären, muss man zuerst den klassischen Algorithmus zur Lösung des Problems betrachten. Dieser heißt Forward Kinematics und beschreibt, dass um den Effektor, also das Ende des Roboterarms, an einem Punkt zu verhindern, die Achsen einzeln dorthin verstellt werden. Zuerst wird also die erste Achse bewegt, dann die zweite usw. Man kann dem Roboterarm also keinen Punkt vorgeben, an den er gehen soll, sondern nur die Winkel, die die Achsen haben sollen. Bei der inversen Kinematik, wie der Name schon sagt, wird dieser Prozess umgekehrt. Anstatt Winkel für die

Achsen einzugeben und einen Punkt für den Effektor zu erhalten, gebe ich bei der inversen Kinematik einen Punkt ein und erhalte die Winkel der Achsen. (Q18)

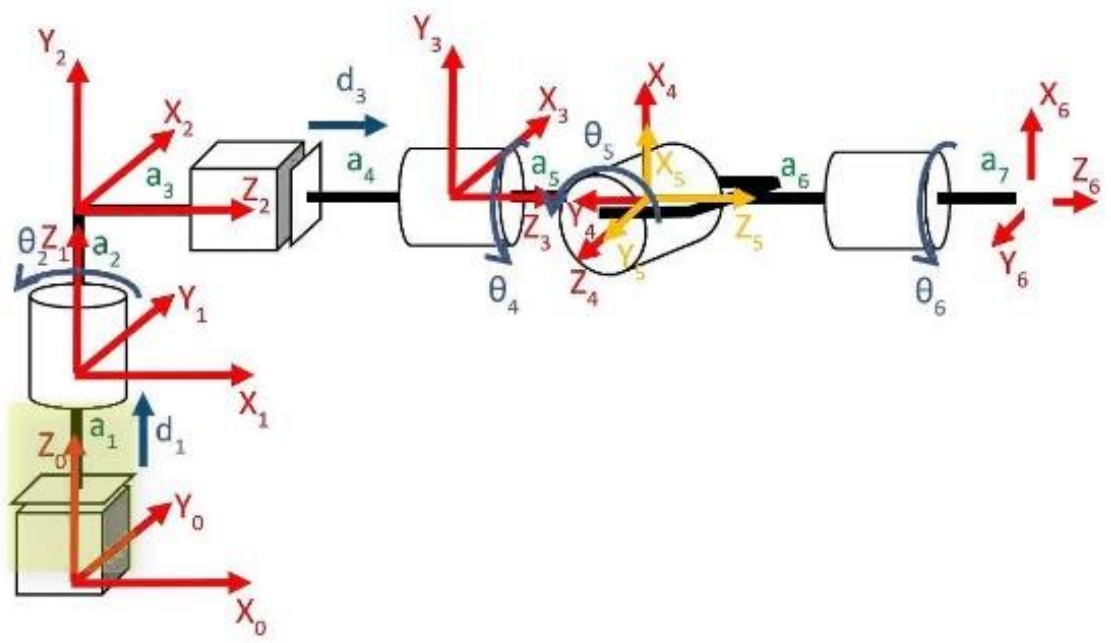


Abbildung: 40

Für verschiedene Achskonfigurationen werden die IK Solver (Skript zur Berechnung der inversen Kinematik) schnell sehr komplex, wie das Beispiel in Abbildung 40 zeigt. Deshalb möchte ich den Aufbau des Roboterarms noch einmal überarbeiten. Die einfachsten für Software verfügbaren IK Solver, sind für Achsen, die alle in die gleiche Richtung zeigen und zusammen eine beliebig lange Kette bilden. Diese werden auch IK Fabric genannt.

Um später eine IK-Fabric einsetzen zu können, entferne ich eine Achse des Roboters, um den Anforderungen gerecht zu werden. Diese lasse ich dann ausdrucken und die erste Version des Roboterarms ist fertig.

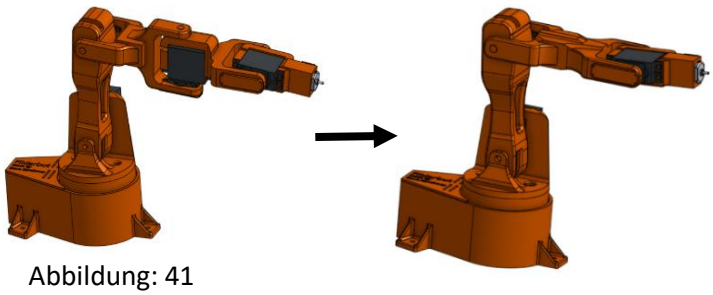


Abbildung: 41

Mit der Version, welche in Abblidung 42 zu sehen ist, konnte ich weitere Tests durchführen und die Software weiterentwickeln. Allerdings gab es einige Probleme mit dem Roboterarm. Ich hatte mich für den Servomotor anstelle des genaueren Schrittmotors entschieden, weil er einfacher zu handhaben ist und ich dachte, dass die Genauigkeit ausreichen würde. Aber ich habe mich getäuscht, denn neben der mangelnden Genauigkeit gibt es noch andere Probleme. Bei der Signalübertragung sende ich viele Signale auf einmal, und die Servomotoren können anscheinend nicht damit umgehen, dass sie, bevor sie sich in die eine Richtung bewegen, schon ein Signal für die andere Richtung bekommen haben. Natürlich sollen sie in der Lage sein, die Richtung zu ändern, aber das geschieht in viel kürzeren Zeitabständen, und deshalb stottern sie.

Abbildung: 42

Die Tatsache, dass ich in der Software zu viele Signale gesendet habe, ist einer der Gründe, warum es weniger geworden ist. Aber das Stottern wird zusätzlich dadurch verursacht, dass der Servo Motor immer mit maximaler Geschwindigkeit in eine Position fahren will. Generell wird auch jedes Ruckeln vom Motor auf die Achsen übertragen, weil ich keine Übersetzung auf Zahnräder habe. Eine Übersetzung auf Zahnräder ist aber nur begrenzt möglich, weil sie sich nur um 180° drehen können, was bei einer Übersetzung bedeuten würde, dass ich den maximalen Winkel noch weiter verkleinern würde.

Als ich diese Probleme analysierte, beschloss ich, einen neuen Roboterarm zu entwickeln, der mit Schrittmotoren angetrieben wird. Dafür brauche ich leistungsfähigere Elektronik, die auch mehr kostet als die alte. Um die Bauteile für meinen Roboterarm günstig zu bekommen, habe ich mich an die Firma Simac gewandt, die viele der benötigten Elektronikteile verkauft und mich beraten konnte. Um finanzielle Unterstützung zu bekommen, brauchte ich einen Sponsor. Als Kandidat für einen Sponsor sah ich das Katholische Karl-Leisner Krankenhaus in Kleve, in dem der Chirurg Dr. Bertrams arbeitet. Um dem Krankenhaus das Projekt vorzustellen, habe ich eine Broschüre für mein Projekt erstellt und diese dem Katholischen Karl-Leisner Krankenhaus in Kleve vorgelegt. Sie haben zugestimmt mich bei meinem Projekt finanziell zu unterstützen.

Um genau zu wissen, welche Teile ich brauche, muss ich erst einmal wissen, wie der neue Roboterarm aussehen soll. Dafür suche ich mir wieder ein Modell aus, mit dem ich weiterarbeiten kann. Ich habe mich für den Roboterarm von RoTechnic (Q19) entschieden, da die Bewegung des Roboters in den Videos sehr geschmeidig aussieht und er den Zusammenbau und die Funktionsweise in seinen Videos ausführlich erklärt. Außerdem finde ich das Aussehen des Roboterarms passend für einen chirurgischen Roboterarm, wenn ich die Farbe von gelb auf weiß ändere. Er hat den Roboterarm so gebaut, dass er eine Kamera präzise steuern kann und hat am Ende des Roboterarms eine Art Motor-Gimbal mit Motoren gebaut. Diesen kann ich abnehmen und durch eine Halterung für einen Gleichstrommotor ersetzen.

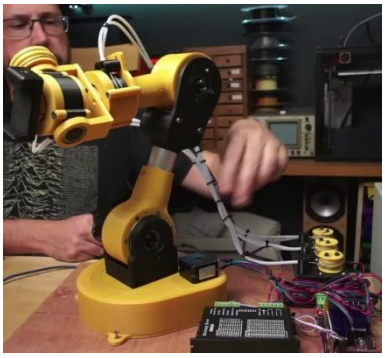


Abbildung: 43

Zunächst zur Funktion des Roboterarms, denn dieser funktioniert noch einmal ganz anders als ein Roboter mit Servomotoren. Der Roboterarm von RoTechnic verwendet Getriebe, um die Motoren stärker zu machen, da Schrittmotoren relativ schwach sind. Er benutzt aber keine normalen Zahnräder, da diese zu groß wären, um ein effektiv starkes Getriebe zu haben. Er verwendet sogenannte Cycloidal Drive, die viel kleiner sind und hier eine Übersetzung von 1:20 erreichen. Allerdings sind sie aufwendig zu konstruieren und zu montieren. Eine wichtige Eigenschaft eines Cycloidal Drive ist seine Fähigkeit, ein hohes Drehmoment bei relativ kleinen Abmessungen zu übertragen. Das macht sie ideal für Anwendungen, bei denen Platzbedarf und Gewicht eine wichtige Rolle spielen, wie bei diesem Roboterarm. (Q21)

Abbildung: 44

Ein Cycloidal Drive funktioniert anders als ein normales Zahnradgetriebe. Das Funktionsprinzip eines Cycloidal Drive beruht darauf, dass die Bewegung der äußeren Scheibe in eine Drehbewegung des inneren Zahnrades umgesetzt wird. Dies geschieht dadurch, dass die Zwischenräume der äußeren Scheibe in die Zähne des inneren Zahnrades eingreifen und so eine Übertragung von Drehmoment und Drehzahl ermöglichen. (Q20)

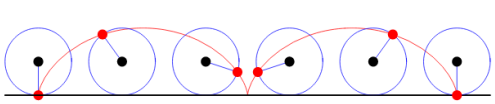


Abbildung: 45

Der Hauptunterschied zwischen einem Cycloidal Drive und anderen Getriebearten besteht darin, dass die Zähne der äußeren Scheibe nicht gleichmäßig verteilt sind, sondern eine spezielle, sich wiederholende Form aufweisen, die als "Epizykloidenkurve" bezeichnet wird. Diese Zahnform ermöglicht es, dass die äußere Scheibe eine größere Anzahl von Zähnen als das innere Zahnrad haben kann, was zu einem höheren Übersetzungsverhältnis führt. (Q21)

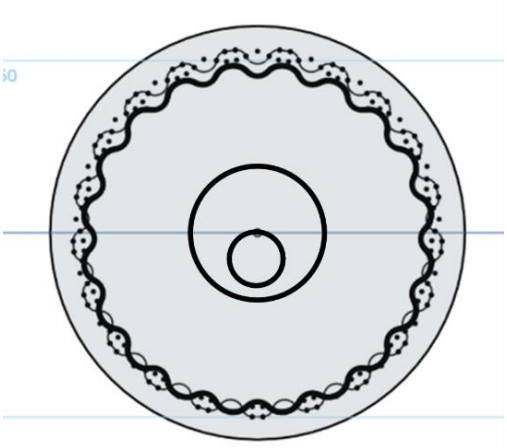


Abbildung: 46

Die Drehmomentübertragung beim Cycloidal Drive erfolgt über die sogenannten Roller. Bei dieser Ausführung des Cycloidal Drive befinden sich in der Scheibe 4 Roller. Diese Rollen sind Zylinder, die in den Löchern im Inneren des Zahnrads laufen. Durch die Bewegung der äußeren Scheibe werden die Rollen entlang der Epizykloidenkurve geführt, wodurch das innere Zahnrad gedreht und das Drehmoment auf die Abtriebswelle übertragen wird.

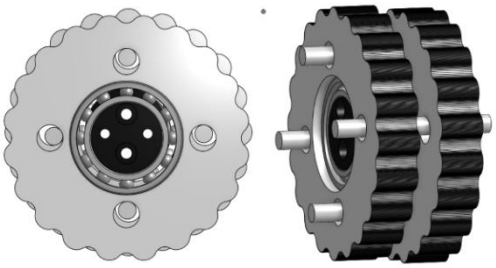


Abbildung: 47

Hier werden zwei Scheiben verwendet, die mit einem dezentrierten Verbundstück verbunden sind. Der Schrittmotor treibt diesen an, wodurch die Zahnräder in entgegengesetzter Richtung die Epizykloidenkurve durchlaufen. Die Zylinder der Scheiben halten die Zahnräder an Ort und Stelle, was schließlich zu einer Drehung des äußeren Gehäuses führt. Dies erklärt, wie der Zykloidenantrieb für die meisten Achsen des Roboterarms funktioniert.

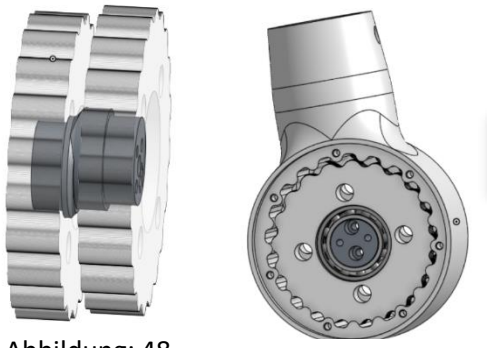


Abbildung: 48

Das Getriebe wird dann so zusammengebaut, dass zunächst der Schrittmotor an einer Hälfte des äußeren Gehäuses montiert wird, an dem die Zylinder für die Scheiben befestigt werden. Dann wird das Gehäuse mit den Scheiben auf die linke Hälfte montiert, in der sich die Scheiben mit den Exzenterverbindungsstücken befinden. Darauf wird eine Haube und die andere Hälfte des Gehäuses montiert.

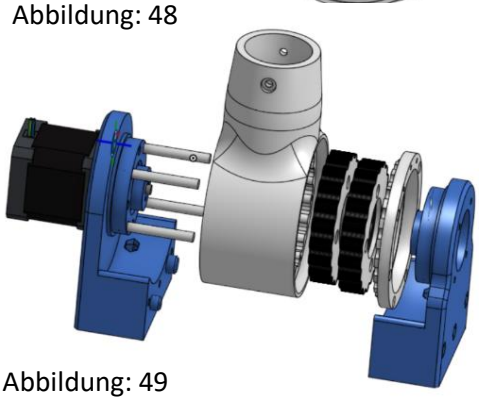


Abbildung: 49

Die Basis verwendet ebenfalls einen Cycloidal Drive, der jedoch anders aufgebaut ist. Die Basis ist ein exzentrischer Cycloidal Drive.

Ein exzentrischer Cycloidal Drive besteht aus zwei Hauptteilen: einem exzentrischen Wellenantrieb und einem Rotor. Das kreisförmige Gehäuse hat eine unregelmäßig geformte Innenwand, die als Stator bezeichnet wird.

Der Exzenterantrieb besteht aus einem kleinen Kreis, der exzentrisch zu einem größeren Kreis angeordnet ist. Dieser kleine Kreis wird Exzenter genannt und dreht sich um die größere Kreisfläche. Der Zahnkranz, auch Rotor genannt, ist in das Gehäuse eingesetzt und hat eine Innenverzahnung, die mit den Zähnen des Stators kämmt. Wenn sich der Exzenter dreht, verschiebt er die Zähne des Rotors, wodurch dieser in Drehung versetzt wird. Um eine noch größere Fläche zwischen Exzenter und Rotor zu erhalten, wird die Zykloidenkurve exzentrisch verschoben.

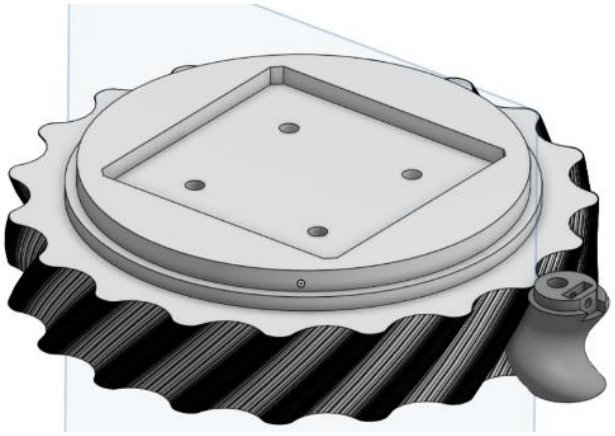


Abbildung: 50

Der Exzenter hat eine Bohrung, in der die Welle des Schrittmotors befestigt wird. Die Welle wird mit einer Mutter und einer Schraube an der flachen Seite der Welle befestigt, um sicherzustellen, dass der Schrittmotor den Welleneingang nicht abnutzt und den Exzenter immer dreht. Auf diese Weise werden auch die anderen Schrittmotoren befestigt.

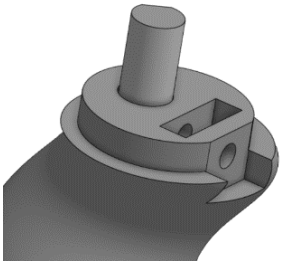


Abbildung: 51

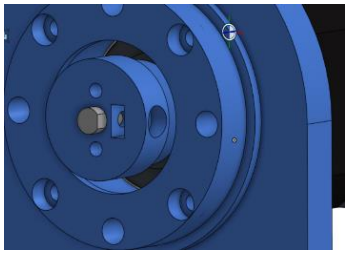


Abbildung: 52

Der Exzenter und der Rotor wurden so gezeichnet, dass kein Spalt zwischen ihnen ist. Beim 3D-Drucken stellte ich fest, dass die Ungenauigkeit des 3D-Druckers dazu führte, dass das Zahnrad zu stark hackte und die exzentrische Oberfläche winzige Abweichungen aufwies, die wahrscheinlich zu Reibung führten. Um das Problem zu lösen, habe ich die Außenfläche des großen Rotors verkleinert. Dabei habe ich den Fehler gemacht, den Rotor in alle Richtungen gleichmäßig zu skalieren. Dadurch gibt es unregelmäßig immer noch Stellen ohne Zwischenraum. Als ich den Fehler erkannte, warum das Zahnrad immer noch hackte, skalierte ich den Rotor entlang der x- und y-Achse und die z-Achse blieb gleich, wodurch das Problem gelöst wurde.

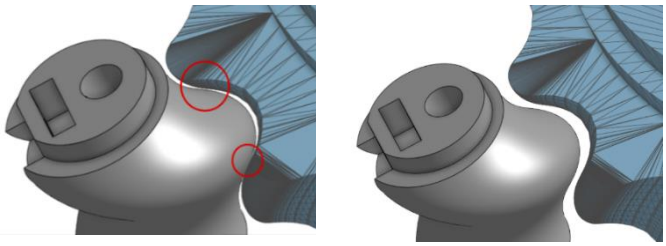


Abbildung: 53

Das Zahnrad ist in ein Gehäuse eingesetzt und der Exzenter wird durch einen Schrittmotor gedreht. Die erste Achse wird von unten auf das Gehäuse geschraubt und darauf wird eine Verkleidung verschruabt.

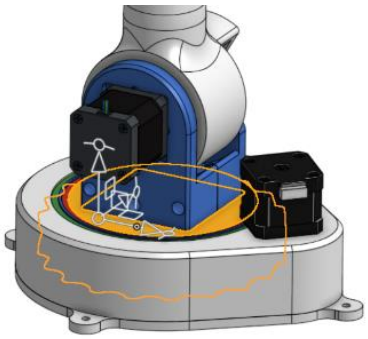


Abbildung: 54

Die folgenden Achsen sind durch Stangen miteinander verbunden, die mit Löchern für die Befestigung durch Schrauben versehen sind.

Den vorderen Teil des Roboterarms habe ich durch eine Halterung für einen Gleichstrommotor ersetzt. Zuerst kommt wieder eine Steckverbindung mit Verschraubungen, um sie mit der anderen Achse verbinden zu können. An diesem Teil sind auf beiden Seiten zwei Halterungen angeschraubt, in denen Kugellager befestigt sind, mit denen sich die DC-Motorhalterung drehen kann.

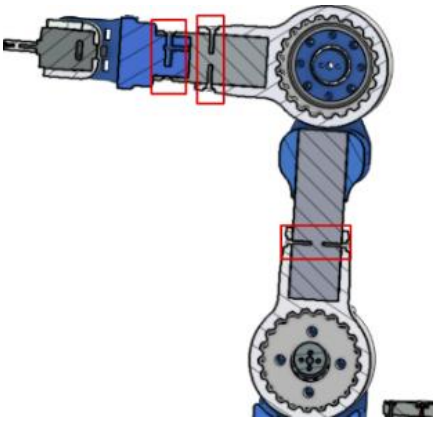


Abbildung: 55

An der linken Seite der Halterung ist ein kleiner Schrittmotor befestigt, der die Drehung der Halterung bewirkt. Die meisten Modelle von Gleichstrommotoren haben keinen Bohrer, deshalb habe ich einen Aufsatz auf den Gleichstrommotor montiert.

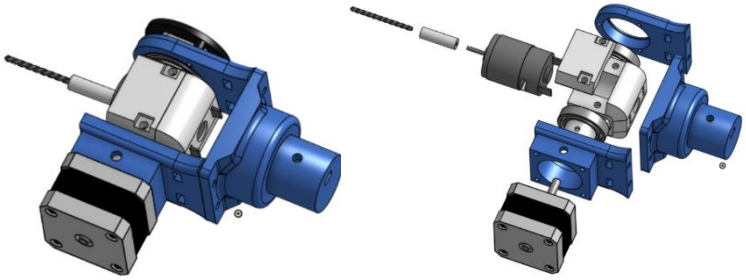


Abbildung: 56

So wird die Konstruktion des Roboterarms sichtbar. Jetzt kann ich mich um die Beschaffung der Elektronik, der Kugellager und anderer Materialien kümmern. Zuerst musste ich mich mit der Ansteuerung eines Schrittmotors vertraut machen. Ich entschied mich für einen NEMA 17 Schrittmotor. NEMA bedeutet National Electrical Manufacturers Association. Das bedeutet, dass die Maße in Kategorien genormt sind und 17 steht für ein genormtes Verhältnis, welches in Zoll angegeben wird. Also NEMA 17 hat eine Montagefläche von 17x17 inch. (Q16.2)

Um einen NEMA 17 Schrittmotor mit einem Arduino zu steuern, werden folgende Komponenten benötigt:

1. NEMA 17 Schrittmotor: Der NEMA 17 Schrittmotor ist ein weit verbreiteter Schrittmotor, der für eine Vielzahl von Anwendungen eingesetzt wird. Er hat vier Drähte, die zur Steuerung des Motors verwendet werden. Diese führen zu den Spulen des Schrittmotors.
2. Arduino: Ein Arduino ist eine Mikrocontroller-Plattform, die für eine Vielzahl von Anwendungen eingesetzt werden kann. Er kann über eine einfache USB-Schnittstelle mit einem Computer oder einem anderen Gerät kommunizieren.
3. Treibermodul: Ein Treibermodul wird benötigt, um den Schrittmotor zu steuern. Es gibt verschiedene Arten von Treibermodulen, die für den NEMA 17 Schrittmotor verwendet werden können. Ich habe mich für den MKS TMC2209 Schrittmotortreiber entschieden. Dieser kann eine Schrittweite von bis zu 1/256 Mikroschritten erreichen und ist damit sehr präzise. Außerdem verfügt er über eine Stallfunktion, bei der der Schrittmotor erkennt, wenn er blockiert ist und stoppt, anstatt weiterzulaufen. Diese Funktion ist nützlich, um den Schrittmotor zu testen. Das Treibermodul steuert die Spulen des Schrittmotors und die Polarität. Außerdem kann man die Stromzufuhr begrenzen, indem man die Schraube am Treiber dreht. Dazu misst man den Prüfstrom vref, der beim Drehen der Schraube zwischen dem GND-Eingang und der Schraube fließt. Dieser Stromfluss wird mit einem Multimeter gemessen, während die Ampereschraube mit einem Schraubendreher eingestellt wird. Diese Einstellung ist nützlich, um den Strom an Motoren anzupassen, die einen niedrigeren Maximalstrom haben. (Q22.1)

Abbildung: 57

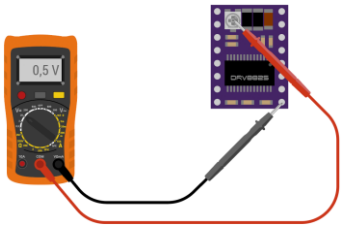


Abbildung: 58

4. Zusätzlich verwende ich ein CNC-Shield, an das mehrere Schrittmotortreibermodule angeschlossen werden können und das direkt auf den Arduino gesteckt werden kann. Das macht die Verkabelung viel einfacher, da ich keine Kabel zwischen den Treibern habe. Außerdem kann man über einen Eingang direkt die benötigte Stromversorgung anschließen. (Q22.2)

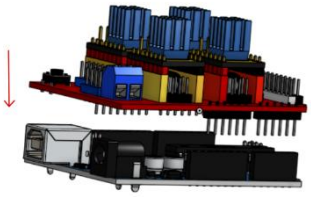


Abbildung: 59

5. Stromversorgung: Der Schrittmotor benötigt zum Betrieb eine bestimmte Spannung und Stromstärke die von den spezifischen Anforderungen des Motors und des Treibermoduls abhängen. Für NEMA 17 ist es normalerweise 12V und abhängig von der Größe des Schrittmotors liegt die Stromstärke für einzelne Schrittmotoren zwischen 1A und 2A. Ich benutze verschiedene Schrittmotorgrößen, um z.B. am Ende des Roboterarms Gewicht zu sparen und an anderer Stelle mehr Kraft und Gewicht für die Stabilität zu haben. Ich werde einen Schrittmotor mit 2A und drei mit 1,5A verwenden. Zusätzlich mit einem DC-Motor, der etwa 2A verbraucht, bin ich bei einem Gesamtstromverbrauch von 8,5A. Das kann sich während des Betriebs noch etwas ändern. Ich habe bei roboter-bausatz.de ein Schaltnetzteil gefunden, das 12v und 20A liefert, was mehr als ausreichend sein sollte. (Q22.3)

Abbildung: 60

Für Haushaltsgeräte gibt es entweder direkt ein passendes Netzteil dabei oder wie bei Ladegeräten für Smartphones ist die benutzte Spannung standardisiert auf 5v. Wenn man aber eine Stromversorgung außerhalb von normalen Haushaltsgeräten benötigt, dann verwendet man Schaltnetzteile, die eine bestimmte Spannung und Stromstärke haben. Das Netzteil, für das ich mich entschieden habe, hat 3 Ausgänge und einen Eingang. Die Ausgänge sind unterteilt in COM für den Minuspol und +V für den Pluspol, an die man die entsprechenden Kabel anschließt. Der Eingang erfolgt über die 3 Pins L, N und PE.

L ist der Lastleiter. Er ist normalerweise schwarz oder braun. Über den Lastleiter gelangt der Strom in das Gerät. Man erkennt ihn daran, dass er als einziger einen Phasenprüfer zum Leuchten bringt. Er ist nie blau oder gelbgrün. Hier ist es braun. (Q23)

N ist der Neutralleiter. Über ihn fließt normalerweise der Strom zurück. Er steht nur dann unter Spannung, wenn er mit dem Leiter L verbunden ist, z. B. wenn eine Lampe zwischen die beiden Leiter geklemmt wird. Diese Leitung ist immer blau. (Q23)

PE ist der sogenannte Schutzleiter, die Erde. Wenn Strom über den Schutzleiter fließt, löst normalerweise die Sicherung aus. Strom auf dem Schutzleiter bedeutet, dass das Gehäuse unter Spannung stehen könnte. Das Kabel, das zum Schutzleiter gehört, ist immer gelb-grün gestreift. So kann man es sofort erkennen. (Q23)

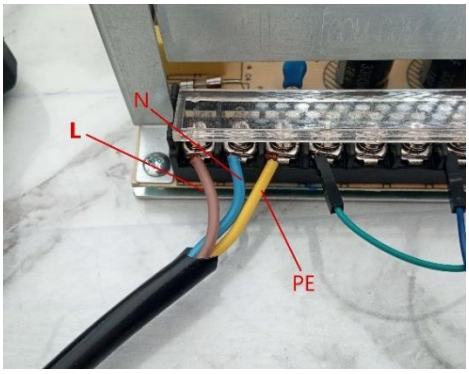


Abbildung: 61

Diese sind in Netzkabeln enthalten, die in die Steckdose gesteckt werden können. Das Schaltnetzteil transformiert den Strom der Steckdose von 230V und 16 Ampere auf 12V und 20 Ampere. Das Schaltnetzteil muss auf den Eingangsstrom der Steckdose eingestellt werden, was durch einen Schalter, der zwischen 110V und 230V wechselt, geschehen kann. 110V ist die Standard-Steckdosenspannung in Nordamerika. Wird die falsche Eingangsspannung eingestellt, kann dies zu Schäden am Transformator oder zu unerwünschten Ausgangsspannungen oder -strömen führen.

Für die Schrittmotoren werden die Treibermodule und das CNC-Shield verwendet. Den Gleichstrommotor steuere ich mit einer einfachen H-Brücke, die Signale von einem anderen Arduino erhält. Dazu verwende ich eine L298N H-Brücke. (Q24)

Die L298N H-Brücke ist eine integrierte Schaltung, die zur Steuerung von Gleichstrommotoren verwendet wird. Sie ist eine bidirektionale Gleichstrom-H-Brücke. Die Funktionsweise der L298N H-Brücke kann in drei Schritten erklärt werden:

Steuersignal: Zunächst muss ein Steuersignal an die H-Brücke angelegt werden. Dies geschieht durch den Anschluss der Steuereingänge an einen Mikrocontroller, in diesem Fall einen Arduino, der digitale Signale erzeugen kann. Jeder Motor benötigt mindestens zwei Eingänge: einen Eingang zur Steuerung der Drehrichtung und einen Eingang zur Steuerung der Geschwindigkeit. (Q24)

Richtungssteuerung: Die H-Brücke ermöglicht die Umkehrung der Drehrichtung eines Motors durch Umkehrung der Stromrichtung. Dies wird durch die

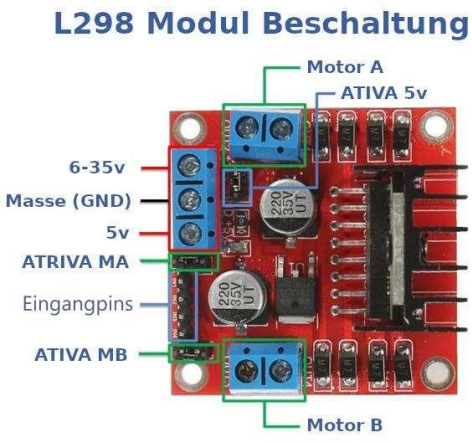


Abbildung: 62

Verbindung von zwei gegenüberliegenden Transistoren (NPN und PNP) erreicht. Wenn der NPN-Transistor leitend ist, fließt der Strom vom positiven Versorgungsanschluss des Motors nach Masse. Wenn der PNP-Transistor leitend ist, fließt der Strom vom negativen Versorgungsanschluss des Motors zur positiven Versorgung. Durch die Ansteuerung der Transistoren können sich die Motoren in beide Richtungen drehen.

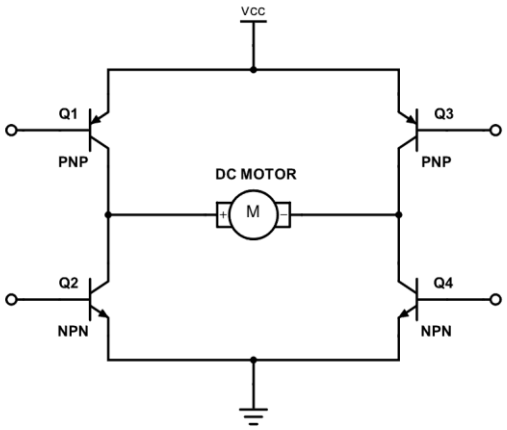


Abbildung: 63

Drehzahlregelung: Die Drehzahl des Motors wird durch Pulsweitenmodulation (PWM) geregelt. Das bedeutet, dass das Steuersignal in kurzen Impulsen an den Motor gesendet wird, wobei die Länge des Impulses die Geschwindigkeit des Motors bestimmt. Durch Veränderung der Pulsbreite kann die Drehzahl des Motors geregelt werden.

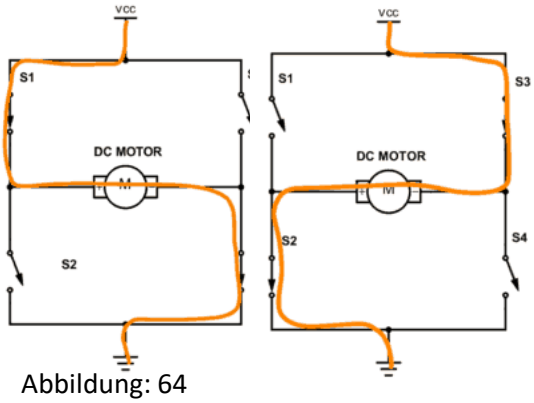


Abbildung: 64

Die H-Brücke L298N ist ein in der Robotik und Automatisierungstechnik häufig eingesetzter Baustein, der die Motorsteuerung einfach und effizient macht. Das IC kann einen maximalen Strom von bis zu 2A pro Kanal und eine maximale Spannung von 5V bis 35V verarbeiten. Bei einer Eingangsspannung über 12V sollte der Ativa 5V Jumper entfernt und eine zusätzliche 5V Stromquelle an den entsprechenden Pin angeschlossen werden. Dadurch wird die Spannung für die Logik und die Motoren getrennt und die Schaltungen geschützt. (Q25)

Der gesamte Schaltplan des Projektes sieht wie folgt aus:

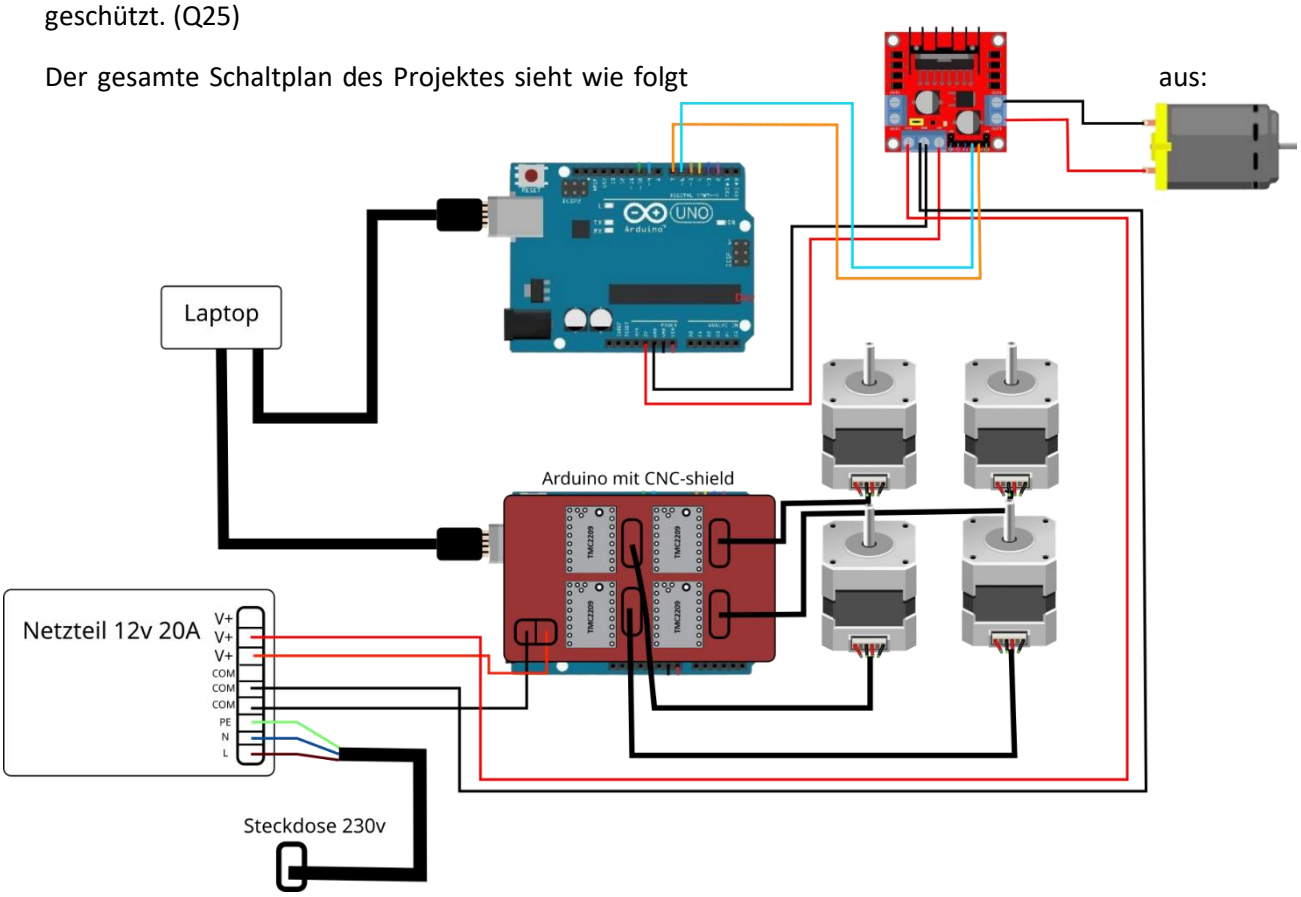


Abbildung: 65

Die Datenübertragung vom Arduino zum CNC-Shield und damit zu den Treibermodulen ist anders als bei den Servomotoren. Man kann das PWM-Signal für den Schrittmotor mit Hilfe einer for-Schleife manuell selbst erzeugen. Dadurch wird ein PWM-Signal erzeugt, das die Geschwindigkeit und Richtung des Schrittmotors bestimmt.

```
const int StepX = 2;
const int DirX = 5;


void setup() {
  pinMode(StepX,OUTPUT);
  pinMode(DirX,OUTPUT);
}

void loop() {
 digitalWrite(DirX, HIGH); // set direction, HIGH for clockwise, LOW for anticlockwise

 for(int x = 0; x<200; x++) { // loop for 200 steps
  digitalWrite(StepX,HIGH);
  delayMicroseconds(500);
  digitalWrite(StepX,LOW);
  delayMicroseconds(500);
 }
 delay(1000); // delay for 1 second// delay for 1 second

}
```

Abbildung: 66

Um die Arbeit mit Schrittmotoren zu erleichtern, kann die Bibliothek AccelStepper hinzugefügt werden. Damit können Objekte für Schrittmotoren wie bei Servomotoren erstellt werden. Für die Objekte gibt es drei Parameter, die eingegeben werden müssen. Der erste beschreibt, ob es sich um einen bipolaren oder unipolaren Schrittmotor handelt. Der NEMA 17 ist ein bipolarer Schrittmotor. Dafür wird die 1 eingegeben. Die nächsten beiden Parameter sind die 2 Eingangspins, einmal der Step Pin und der Direction Pin. Der Step Pin sendet die Geschwindigkeit in Schritten und der Direction Pin die Richtung in die der Schrittmotor laufen soll. Das CNC-Shield dient als fertige Verdrahtung für die Treibermodule, bei denen die Pins für die 4 Schrittmotoren vorgegeben sind. Die Pins für die 4 Schrittmotoren sind wie folgt: 2, 5 | 3, 6 | 4, 7 | 12, 13. (Q26)

sketch_mar30a.ino

```
#include <AccelStepper.h>

AccelStepper Xaxis(1, 2, 5); // pin 2 = step, pin 5 = direction
AccelStepper Yaxis(1, 3, 6); // pin 3 = step, pin 6 = direction
AccelStepper Zaxis(1, 4, 7);  // pin 4 = step, pin 7 = direction
AccelStepper Aaxis(1, 12, 13); // pin 12 = step, pin 13 = direction

const byte enablePin = 8;

void setup() {

  pinMode(enablePin, OUTPUT);
  digitalWrite(enablePin, LOW);

  Xaxis.setMaxSpeed(1000);
  Xaxis.setAcceleration(2000);
}

void loop() {
  Xaxis.moveTo(angleSteps(val1));
  Xaxis.run();
}

float angleSteps(float angle){
  return angle* 400/90;
}
```

Abbildung: 67

Im Setup() muss zuerst der Enable Pin, der meistens auf 8 liegt, aktiviert werden, damit das Board aktiviert wird. Danach muss für jeden Motor die maximale Geschwindigkeit und Beschleunigung mit setMaxSpeed() und setAcceleration() eingestellt werden. In loop() kann mit verschiedenen Funktionen die Position des Schrittmotors eingestellt werden. Mit move() wird der Schrittmotor relativ zu seiner aktuellen Position um die angegebenen Schritte bewegt. Mit moveTo() kann man stattdessen globale Positionen verwenden, so dass der Schrittmotor auf die Position 300 Schritte oder 0 Schritte fährt. Damit wird dem Objekt die Geschwindigkeit gegeben und mit run() wird die Funktion wiederholt ausgeführt. Um statt mit Steps mit Winkeln arbeiten zu können, habe ich eine Funktion implementiert, die die eingegebenen Winkel in Steps umwandelt. Das Verhältnis zwischen Winkel und Schritt ist 400/90, d.h. 400 Schritte entsprechen 90°. Statt run() ständig auszuführen, kann man auch mit distanceTogo() abfragen, ob der Schrittmotor schon an der Stelle ist. Das führt aber in der Praxis dazu,

```
if (Xaxis.distanceToGo() != 0){
  Xaxis.run();
}
```

Abbildung: 68

dass die Motoren stottern, wenn sie immer wieder kleine Abstände zwischen den Schritten bekommen. Daher ist die kontinuierliche Abfrage besser geeignet. (Q26)

Ich benutze diese Funktion, weil ich wie bei den Servomotoren dem Schrittmotor sagen kann, welche Position er einnehmen soll. Es gibt einen deutlichen Unterschied. Der Arduino merkt sich zwar während des Betriebs welche Position der Schrittmotor gerade einnimmt, aber beim Einschalten des Arduinos hat er keine Ahnung wie die Schrittmotoren ausgerichtet sind. Deshalb muss beim Ein- und Ausschalten des Arduinos immer die gleiche Position der Schrittmotoren gewählt werden, damit die Schrittmotoren immer richtig kalibriert sind.

Damit habe ich die Schrittmotoren wie die Servomotoren eingestellt und kann sie in das vorherige Skript für die Datenübertragung eintragen.

```
if (receivedChars[0] == 'a'){
  val1 = temp.toInt();
  Xaxis.moveTo(angleSteps(val1));
  Serial.println(val1);
}
```

Abbildung: 69

Die Ansteuerung der H-Brücke L298N ist relativ einfach. Man braucht nur zwei Ausgangspins, die man als solche deklariert. Ich habe die Pins 6 und 7 genommen.

```
11  int dcspeed = 0;
12
13  //--------------
14
15  int inp1 = 6;
16  int inp2 = 7;
17
18  void setup() {
19    Serial.begin(9600);
20    Serial.println("<Arduino is ready>");
21
22    pinMode(inp1, OUTPUT);
23    pinMode(inp2, OUTPUT);
24  }
```

Abbildung: 70

In Loop können dann die benötigten Signale gesendet werden. Mit analogWrite() kann ein PWM-Signal mit der Periodendauer dcspeed an den ersten Pin gesendet werden. Die Periodendauer kann von 0-255 gewählt werden und bestimmt somit die Geschwindigkeit des DC-Motors. Der Pin wird als Digital Pin mit digitalWrite() benutzt. Mit dem Wert HIGH wird die Stromverbindung hergestellt, mit LOW abgebrochen und der Motor hört auf sich zu drehen. Das kann ich dann mit dem Skript von Unity machen, indem ich die Buchstaben von „o“ und „f“ abfrage.

```
if (receivedChars[0] == 'o'){
  dcspeed = temp.toInt();
  analogWrite(inp1, dcspeed);
  digitalWrite(inp2, HIGH);
  Serial.println(dcspeed);
}

if (receivedChars[0] == 'f'){
  dcspeed = temp.toInt();
  analogWrite(inp1, dcspeed);
  digitalWrite(inp2, LOW);
  Serial.println("turned off");
}
```

Abbildung: 71

Benutzeroberfläche und Pathfill Algorithmus:

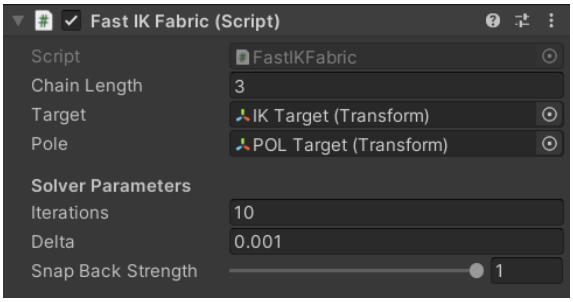


Abbildung: 72

Die Software muss den Roboterarm als digitalen Zwilling bewegen können. Dazu füge ich ein IK-System in die Software ein. Da es viele Anwendungsmöglichkeiten für IK-Systeme gibt, gibt es bereits viele fertige Vorlagen, die ich verwenden kann, anstatt es von Grund auf selbst zu programmieren. Ich benutze dazu das kostenlose und Open Source Add-On „Fast IK“ (Q27) für Unity. Das Script funktioniert so, dass man es am Effektor, also am Ende der Kette, als Komponente hinzufügt. In den Skriptparametern kann man dann die Länge der Kette eingeben, wie oft pro Sekunde der Solver aktualisiert werden soll, bei Iterationen den Abstand zwischen Target und Effektor mit Delta und Snap Back Strength bestimmt, wie schnell die Kette in die gestreckte Form übergehen soll, wenn das Target zu weit weg ist.

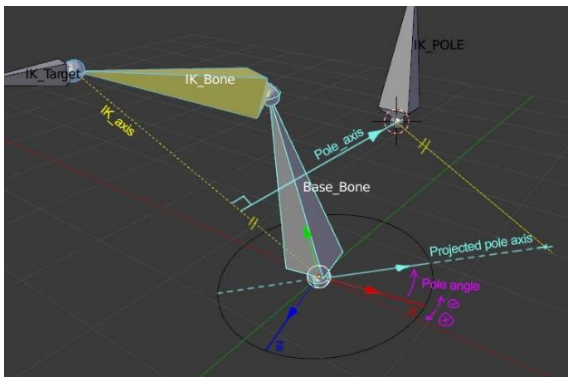


Abbildung: 73

Das IK Target und das Pole Target sind die Objekte, mit denen das IK Fabric bewegt wird. Das Target ist der Punkt, zu dem sich der Effektor bewegen soll. Somit weiß der Solver wohin er sich bewegen soll und passt die Winkel der Achsen durch komplizierte Matrizenmultiplikation, auf die hier nicht weiter eingegangen wird, entsprechend an. Bei diesem IK-Solver gibt es aber immer eine Richtung, in die sozusagen der Ellbogen des Rigs zeigt. Er könnte den Punkt auf vielen verschiedenen Weißen erreichen, weiß aber nicht, in welche Richtung sich das Rig beugen soll. Hier kommt das Pole Target ins Spiel. In die Richtung, in die das Pole Target zeigt, zeigt die Biegung des Rigs. Hier kann man den Grund sehen, warum IK Solver mit mehreren Achsen und Konfigurationen schnell sehr komplex werden können, weil es immer mehr Möglichkeiten gibt, den Punkt zu erreichen und ich das Rig mit immer mehr Bedingungen einschränken muss, damit es einen linearen Weg zum Punkt findet. (Q18.1)

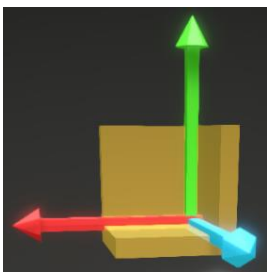


Abbildung: 74

Damit habe ich jetzt ein IK-System. Jetzt muss ich noch Controller für das IK Target programmieren, damit ich es auch in meiner Software bewegen kann. Dazu möchte ich ähnlich wie in Unity und Blender ein Objekt haben, das man anklickt und wenn es ausgewählt ist, zeigt es drei farbige Pfeile, mit denen man es in x, y, z Richtung bewegen kann.

Um die Pfeile umzusetzen, überlege ich mir, wie der Raycast funktionieren muss, um die Pfeile in die entsprechende Richtung bewegen zu können. Dazu soll von der Mausposition ein Raycast gesendet werden. Wenn dieser auf den Pfeil trifft und die Maus gedrückt gehalten wird, soll man den Pfeil entlang einer Ebene bewegen können. Dabei ist die Bewegung auf 2 Achsen beschränkt. Die Achse, die ich für die Position des Pfeils nicht brauche, kann ich ignorieren.

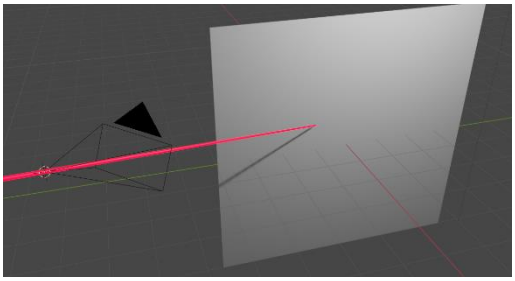


Abbildung: 75

```
Ray ray = Camera.main.ScreenPointToRay(translate.mousePosition);
Plane plane = new Plane(platte, new Vector3(startPos.x + ChangeVector.x, startPos.y + ChangeVector.y, startPos.z + ChangeVector.z));

if (axis == "y")
{
    plane = new Plane(platte, transform.position);
}
float dist = 0;

if (plane.Raycast(ray, out dist))
{
    Vector3 pos = ray.GetPoint(dist);
    raison = pos;

    if (axis == "x")
    {
        tempPos = new Vector3(pos.x, pos.y, pos.z + arrowOffset);
    }
    else if (axis == "y")
    {
        tempPos = new Vector3(transform.parent.position.x, pos.y + arrowOffset, transform.parent.position.z);
    }
    else if (axis == "z")
    {
        tempPos = new Vector3(pos.x + arrowOffset, transform.parent.position.y, transform.parent.position.z);
    }

    transform.parent.position = tempPos;
}
```

Abbildung: 76

Die Umsetzung sieht wie folgt aus. Zuerst wird der Strahl mit der Mausposition als Ursprung erzeugt. Dann wird die Plane erzeugt, die später für den Raycast verwendet wird. Wenn der Raycast die Plane getroffen hat, wird der Treffpunkt mit ray.GetPoint(dist) in der Variabel pos gespeichert. Als nächstes wird abgefragt, in welche Achse der Pfeil bewegt werden soll. Dann wird die neue Position mit der Variabel pos in eine Variabel tempPos zwischengespeichert, indem ein neuer Vektor gebildet wird, der die ursprünglichen Koordinaten des Pfeils enthält und die gewünschte Achse durch pos.x/y/z ersetzt wird. Die Variabel namens arrowOffset wird hinzugefügt, um den Punkt verschieben zu können, an dem der Pfeil der Maus folgt. Ich habe diesen Wert auf die Pfeilspitze gesetzt.

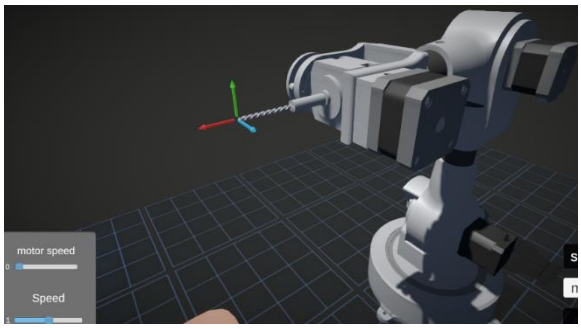


Abbildung: 77

Damit kann ich nun den Roboterarm manuell bewegen, um ihn z.B. an einen Kalibrierungspunkt zu bringen, so dass ich eine digitale Repräsentation habe, die mit den physischen Objekten identisch ist.

Pathfill Algorithmus:

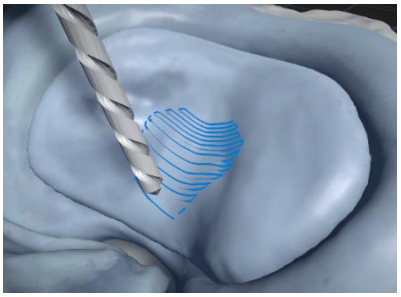


Abbildung: 78

Der Roboterarm soll dann in der Lage sein, eine ausgewählte Fläche am Knie selbstständig zu entfernen. Dazu muss ein komplexer Algorithmus entwickelt werden, der aus einer Fläche einen eindimensionalen Pfad generiert. Dazu habe ich mir bereits existierende Algorithmen angesehen, um eine Vorstellung davon zu bekommen, wie dieser aussehen könnte. Schließlich habe ich meinen eigenen Algorithmus erstellt, dessen Pfad sich an die im Video (Q18.2) angedeutete Methode anlehnt. Die Hilber-Kurve war nicht geeignet, da der Roboter so wenig Kurven wie möglich fahren sollte, da lineare Bewegungen genauer ausgeführt werden können als Richtungsänderungen, weswegen ich die klassische „Snake Kurve“ benutzt habe.

Folgende Aufgaben sind zu lösen:

- Erstellen eines Pfades, der eine einfache Fläche (z. B. ein Viereck) vollständig durchläuft.
- Übertrage diesen Pfad auf alle möglichen Polygone.
- Umwandlung des Pfades in eine 3-dimensionale Projektion

```
float genDirection = 1;
for (int i = 0; i < xSizeAdjusted; i++)
{
    //offsets rows to have a rectangular shape again, when direction is changed
    float offset = 0;
    if (!int.TryParse((i * 0.5f).ToString(), out int value))
    {
        offset = (zSizeAdjusted - 1) * zDensity * densityScale;
    }
    //generates objects and adds them to pathpointsList
    for (int j = 0; j < zSizeAdjusted; j++)
    {
        GameObject pathpoint = (GameObject)Instantiate(prefabPathPoint, spawnParent);
        pathpoint.transform.localPosition = new Vector3(i * xDensity * densityScale + cornerPos1.x, cornerPos1.y, j
        pathpoint.name = "pathpoint" + j;
        pathpointsList.Add(pathpoint);
        pathpoint.GetComponent<Pathpoint>().checkExtent = checkExtent;
        pathpoint.GetComponent<Pathpoint>().checkMask = checkMask;
    }
    //change direction of path every second row so path is connected at ends correctly
    genDirection *= -1;
}
```

Abbildung: 79

Zu diesem Zweck habe ich den Code, welcher in Abbildung 79 zu sehen ist, erstellt. Dieser erzeugt eine Pfadlinie aus einer Liste von Spielobjekten (GameObjects) extrahiert und zu einer Liste von Pfadpunkten (pathpointsList) hinzufügt. Spielobjekte sind Grundlegende Objekte in Unity, die Komponenten besitzen können, wie Skripte und Positions und rotations Informationen. Die Pfadlinie wird durch die Anzahl der Zeilen und Spalten von Spielobjekten definiert, die in einer rechteckigen Formation angeordnet sind. Die Größe und Position jedes Spielobjekts wird durch den Skalierungsfaktor (densityScale) und die Position der Ecken des Rechtecks (cornerPos1) bestimmt.

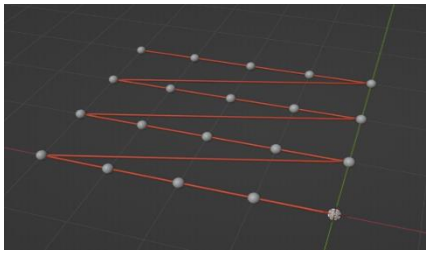


Abbildung: 80

Die Schleife durchläuft jede Zeile (i) der Spielobjekte, und für jede Zeile wird eine verschachtelte Schleife durchlaufen, die jedes Spielobjekt in der aktuellen Zeile (j) erzeugt und zu pathpointsList hinzufügt. Die Position jedes Spielobjekts wird basierend auf der aktuellen Zeile (i), der aktuellen Spalte (j) und der aktuellen Richtung (genDirection) berechnet.

Die Richtung wird durch eine Variable (genDirection) gesteuert, die zu Beginn der Schleife auf 1 gesetzt wird. Für jede gerade Linie wird genDirection auf -1 gesetzt, um die Richtung umzukehren und sicherzustellen, dass die Pfadlinie am Ende verbunden bleibt.

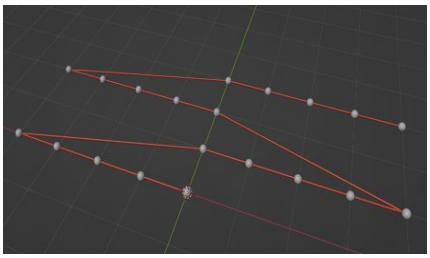


Abbildung: 81

In der Schleife wird auch eine Bedingung verwendet, um sicherzustellen, dass die Spielobjekte in jeder Zeile in einer rechteckigen Form angeordnet werden. Wenn eine ungerade Zeile erreicht wird, wird ein Offset berechnet, um sicherzustellen, dass die Spielobjekte in der nächsten geraden Zeile korrekt angeordnet sind.

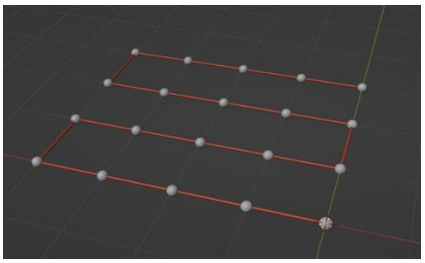


Abbildung: 82

Die Variablen xSizeAdjusted und zSizeAdjusted definieren die Anzahl der Zeilen und Spalten der Spielobjekte in der Pfadlinie, die aus den Variablen xDensity und zDensity berechnet wird. Die Variable prefabPathPoint gibt das Template-Spielobjekt an, das für jeden Pfadpunkt instanziiert wird. spawnParent ist das Elternobjekt, dem alle Pfadpunkte als Kinder hinzugefügt werden.

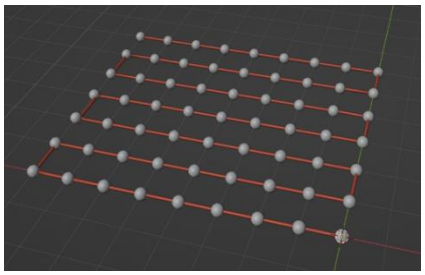


Abbildung: 83

Damit kann ich bereits Objekte entlang eines Pfades erzeugen und mit der Liste abrufen. Die Objekte sind noch kein durchgehender Pfad, sondern nur in Form eines Pfades aneinandergereihte Objekte.

```
if (currentWaypoint == null)
{
    currentWaypoint = waypoints.GetNextWaypoint(null);
}

if (currentWaypoint != null && currentWaypoint != transform && StartFollowPath)
{
    transform.position = Vector3.MoveTowards(transform.position, currentWaypoint.position, 2 *
    if (Vector3.Distance(transform.position, currentWaypoint.position) < distanceThreshold)
    {
        currentWaypoint = waypoints.GetNextWaypoint(currentWaypoint);
    }
}
```

Abbildung: 84

Das Skript in Abbildung 84 bewegt ein Objekt entlang eines Pfades. Zuerst wird geprüft, ob currentWaypoint null ist und wenn ja, wird der nächste Pfadpunkt als Startpunkt gesetzt. Wenn currentWaypoint ungleich Null und StartFollowPath wahr ist, bewegt sich das Objekt zum nächsten Pfadpunkt und prüft, ob es sich dem aktuellen Pfadpunkt nähert (mit distanceThreshold als Schwellwert). Wenn der Schwellwert überschritten wird, wird der nächste Wegpunkt als Ziel gewählt. (Q28)

Es wird geprüft, ob das Ende der Pfadlinie erreicht ist und wenn ja, wird StartFollowPath auf false, returnToStart auf true und der nächste Pfadpunkt auf null gesetzt.

```
if (waypoints.spawnParent.childCount != 0 &&  currentWaypoint == waypoints.spawnParent.GetChild(waypoints.spawnParent.childCount-1))
{
    StartFollowPath = false;
    returnToStart = true;
}
```

Abbildung: 85

Wenn returnToStart wahr ist, wird das Objekt zum Startpunkt zurückbewegt, indem es zur defaultPosition bewegt wird. Wenn das Objekt die Startposition erreicht hat, wird der nächste Pfadpunkt als Startpunkt festgelegt, returnToStart wird auf falsch gesetzt und die Variable DCsender.unlockDcMotor wird auf falsch gesetzt.

```
if (returnToStart)
{
    if (transform.position != defaultPosition)
    {
        transform.position = Vector3.MoveTowards(transform.position, defaultPosition, 2 * manage.movementSpeed * Time.deltaTime);
    }
    else
    {
        currentWaypoint = waypoints.GetNextWaypoint(null);
        returnToStart = false;
        DCsender.unlockDcMotor = false;
    }
}
```

Abbildung: 86

Die Variable DCsender.unlockDcMotor muss während des Durchlaufens des Pfades wahr und am Ende des Pfades falsch sein. Damit kann ich den Gleichstrommotor während des Durchlaufens des Pfades aktivieren.

Damit habe ich einen Pfad, der als Rechteck erzeugt werden kann und durch Anpassung der Punktdichte in der Fläche angepasst werden kann. Um den Pfad an eine beliebige Oberfläche anzupassen, hat jeder PathPoint ein Skript.

```
void Start()
{
    isSpawned = false;
    path = FindObjectOfType<Pathpoints>();

    bool InShape = Physics.CheckBox(transform.position, checkExtent, path.transform.rotation, checkMask);

    if (!InShape && path.alignToShape)
    {
        isSpawned = true;
        path.OutsideShape(this);
    }

}
```

Abbildung: 87

Nachdem das Objekt erzeugt wurde, wird "InShape" mit einem Aufruf von "Physics.CheckBox()" initialisiert. Dieser Aufruf überprüft, ob sich das aktuelle Spielobjekt innerhalb eines definierten Bereichs befindet, der durch den Mittelpunkt des Spielobjekts, die Größe der Box (checkExtent) und die Ausrichtung des Pfads (path) definiert ist. Das Argument "checkMask" gibt eine Schichtmaske an, um zu definieren, welche Kollisionsobjekte in die Berechnung einbezogen werden sollen.

Wenn sich das aktuelle Spielobjekt nicht innerhalb des definierten Bereichs befindet, wird "isSpawned" auf "true" gesetzt und die Methode "OutsideShape()" von "path" aufgerufen, um das Spielobjekt

außerhalb der definierten Form zu löschen. In der Methode Path.OutsideShape() wird das Objekt PathPoint auch aus der Liste pathpointsList entfernt.

Dadurch wird der Pfad an die Form angepasst. Je höher die Auflösung, d.h. je mehr Pfadobjekte, desto genauer wird die Approximation.

```
if (start)
{
    //store rotation/position
    storeRotation = storeRotationView;
    storePosition = storePositionView;

    //set rotation/position to origin
    transform.eulerAngles = Vector3.zero;
    transform.position = Vector3.zero;

    //updateMesh
    selecter.updateMesh(ShapeMesh);

    makeVisible(false);
}
else
{
    //restore rotation
    transform.rotation = storeRotation;
    transform.position = storePosition;

    makeVisible(true);

    for (int i = 0; i < pathpointsList.Count; i++)
    {
        RaycastHit hit;

        if (Physics.Raycast(pathpointsList[i].transform.position, Vector3.down, out hit))
        {
            pathpointsList[i].transform.position = hit.point + new Vector3(0, pathHeight, 0);
        }
    }
}
```

Abbildung: 88

Bisher kann dieser Pfad nur von einer Position aus erstellt werden und ist nicht drehbar.

Anstatt die Erstellung des Pfades komplett anzupassen, lasse ich die Erstellung so wie sie ist und verschiebe den Pfad vor und nach der Erstellung. Dazu habe ich den Code in Abbildung geschrieben. Wenn der Pfad generiert werden soll, wird die Methode mit Start = True aufgerufen, um die aktuelle

```
yield return new WaitUntil(() => transform.eulerAngles == Vector3.zero && transform.position == Vector3.zero);

//generates objects
float genDirection = 1;
for (int i = 0; i < xSizeAdjusted; i++)
{
    //offsets rows to have a rectangular shape again, when direction is changed
    float offset = 0;
    if (!int.TryParse((i * 0.5f).ToString(), out int value))
    {
        offset = (zSizeAdjusted - 1) * zDensity * densityScale;
    }
    //generates objects and adds them to pathpointsList
    for (int j = 0; j < zSizeAdjusted; j++)
    {
        GameObject pathpoint = (GameObject)Instantiate(prefabPathPoint, spawnParent);
        pathpoint.transform.localPosition = new Vector3(i * xDensity * densityScale + cornerPos1.x, cornerPos1.y, j * zDensity
        pathpoint.name = "pathpoint" + j;
        pathpointsList.Add(pathpoint);
        pathpoint.GetComponent<Pathpoint>().checkExtent = checkExtent;
        pathpoint.GetComponent<Pathpoint>().checkMask = checkMask;
    }
    //change direction of path every second row so path is connected at ends correctly
    genDirection *= -1;
}

yield return new WaitUntil(() => allSpawned);
```

Abbildung: 89

Position und Rotation in den Variablen storePosition und storeRotation zu speichern. Danach werden Rotation und Position auf 0 gesetzt. Danach wird der Pfadgenerierungscode ausgeführt und die Methode erneut aufgerufen. Diesmal mit start = false und die gespeicherten Werte von storePosition und storeRotation werden eingelesen. Mit der Methode makeVisible wird der Pfad ausgeblendet bis er generiert und wieder an der richtigen Position angezeigt wird.

Ein Problem, das auftrat, war, dass das Pfadobjekt zu schnell zurückkehrte und die gespeicherte Position und Rotation verwendete, wodurch der Pfad falsch generiert wurde. Das lag daran, dass die beiden Prozesse asynchron ablaufen. Um sicherzustellen, dass die Methoden immer nacheinander ausgeführt werden, habe ich den Code mit waitUntil() => boolscher Wert warten lassen, bis er die Position eingenommen hat. Und nach der Pfadgenerierung wartet er, bis wirklich alle Objekte erzeugt wurden. So kann ich den Pfad zu anderen Positionen bewegen und drehen. Der Code dafür ist in Abbildung 83 zu sehen.

Damit es nicht nur zweidimensional ist, sondern sich an die Oberfläche anpasst, verwende ich am Ende der Pfadgenerierung einen Raycast. Dieser Code durchläuft alle Elemente einer Wegpunktliste (pathpointsList) und korrigiert die Höhe jedes Wegpunktes entsprechend der Oberfläche unter dem Wegpunkt.

In der Schleife wird zunächst eine Variable vom Typ RaycastHit definiert, die verwendet wird, um Informationen über das getroffene Objekt zu speichern. Dann wird ein Raycast von der Position jedes Wegpunktes in Richtung der negativen Y-Achse (d.h. nach unten) durchgeführt. Wenn der Raycast auf ein Objekt trifft, wird die Position des Wegpunktes auf den Punkt des getroffenen Objektes plus einer neuen Vector3-Instanz mit einer Höhenkorrektur (pathHeight) gesetzt, um sicherzustellen, dass der Wegpunkt auf der Oberfläche des Objektes liegt. Wenn der Raycast kein Objekt trifft, bleibt die Position des Wegpunktes unverändert. Das Ergebnis ist eine Anpassung des Weges an die Oberfläche.

```
for (int i = 0; i < pathpointsList.Count; i++)
{
    RaycastHit hit;

    if (Physics.Raycast(pathpointsList[i].transform.position, Vector3.down, out hit))
    {
        pathpointsList[i].transform.position = hit.point + new Vector3(0, pathHeight, 0);
    }
}
```

Abbildung: 90

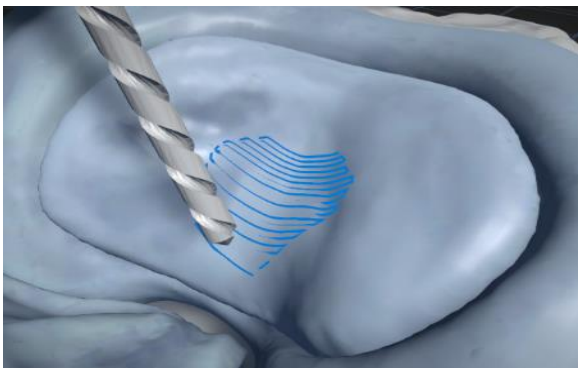


Abbildung: 91

Flächeneditor:

Im Moment passt sich der Pfad nur an eine vordefinierte Fläche an. Ich muss eine Benutzerschnittstelle entwickeln, die es erlaubt, eine Fläche auf dem Knie auszuwählen. Dazu habe ich eine modifizierte Version des Scripts aus diesem Video (Q29) verwendet. Das Skript verwendet ein Raycast, um die Mausposition auf eine Fläche zu projizieren. Mit einem Rechtsklick wird ein Punkt erzeugt.

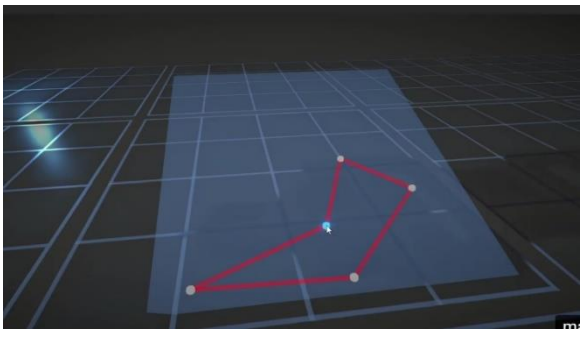


Abbildung: 92

Das gezeigte Skript im Video ist im Prinzip genau was ich für meine Benutzerschnittstelle brauche. Ich kann es aber nicht eins zu eins übernehmen, weil es im UnityEditor läuft und nur als Editor für das erstellen von Flächen während der Entwicklung gedacht ist die später im Programm statisch vorhanden sein sollen. Ich will aber einen Editor entwickeln der im kompillierten Programm funktioniert. Um im UnityEditor zu laufen spricht das Skript eine andere api von Unity an, weswegen ich die Funktionen stark überarbeiten damit es im kompilierten Programm funktioniert.

Der Code in Abbildung 93 definiert eine Methode mit dem Namen "mouseButtonDown", die aufgerufen wird, wenn die linke Maustaste gedrückt wird. Die Methode hat einen Parameter namens "ctx", der den Kontext des Input-Events enthält.

Die Methode prüft zunächst, ob sich die Maus über dem Editierbereich befindet. Ist dies der Fall, wird die Variable "hasChanged" auf "true" gesetzt. Dadurch wird der Pfad neu generiert, da ein Punkt geändert oder hinzugefügt wurde.

```
void mouseButtonDown(InputAction.CallbackContext ctx)
{
    if (hitSurfaceCatcher)
    {
        hasChanged = true;

        if (mouseOverPointIndex == -1)
        {
            GameObject surfacePoint = (GameObject)Instantiate(facePoint, surfaceCatcher)

            if (insertIndex == -1)
            {
                facePoints.Insert(facePoints.Count, surfacePoint);
                selectionIndex = facePoints.Count - 1;
            }
            else
            {
                surfacePoint.transform.SetSiblingIndex(insertIndex + 1);
                facePoints.Insert(insertIndex + 1, surfacePoint);
                selectionIndex = insertIndex + 1;
            }

            surfacePoint.transform.position = mousePosition;
            surfacePoint.transform.localScale = Vector3.one * selectRadius;

            regenerateLines();
        }
        else
        {
            selectionIndex = mouseOverPointIndex;
        }
    }
}
```

Abbildung: 93

Wenn der Index des mit der Maus ausgewählten Punktes "-1" ist, wird ein neues GameObject mit dem Namen "surfacePoint" instanziiert und der Liste "facePoints" hinzugefügt. mouseOverPointIndex ist -1, wenn sich die Maus über keinem anderen Punkt befindet. Das GameObject wird an der Mausposition platziert und eine Skalierung auf den Wert von "selectRadius" gesetzt.

Wenn der Index des ausgewählten Punktes nicht "-1" ist, wird der Index des ausgewählten Punktes auf "selectionIndex" gesetzt. Wenn die linke Maustaste auf einem Punkt gedrückt gehalten wird, kann der Punkt mit der Methode mouseButtonDrag() auf der Oberfläche verschoben werden.

```
void mouseButtonDrag(Vector3 mousePosition)
{
    if (Inputs.mouseLeft.IsPressed() && selectionIndex != -1 && hitSurfaceCatcher)
    {
        hasChanged = true;
        facePoints[selectionIndex].transform.position = mousePosition;
    }
}
```

Abbildung: 94

Die Liste der Punkte kann mit einem LineRenderer visuell verbunden werden. Es können nun Punkte erzeugt werden, die visuell verbunden sind. Die Reihenfolge der Punkte in der Liste bestimmt, wie sie vom LineRenderer verbunden werden. Es ist jedoch nicht möglich, Punkte innerhalb von Linien zu erzeugen.

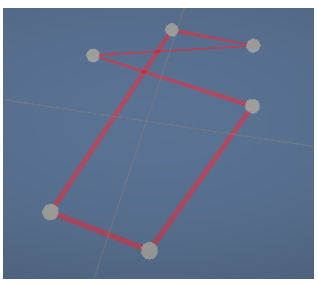


Abbildung: 95

Um dies zu ändern, müssen Linien zwischen den Punkten berechnet werden, um die Entfernung zu bestimmen. Dieses Problem kann mathematisch betrachtet werden. Die Punkte bilden eine Geradengleichung. Die Maus projiziert ihre Position auf einen Punkt auf der Plane. Man muss also die Entfernung von einem Punkt zu einer Geradengleichung berechnen.

Mit der Formel in Abbildung 96 kann der Wert t berechnet werden, der das Verhältnis zwischen den Punkten StartPoint und EndPoint darstellt. Diese Formel ist eine Abwandlung der Formel in Abbildung 97, die für 3-dimensionale Linien funktioniert und mit der ich anstelle des Abstands das Verhältnis t erhalte, mit dem ich den Punkt bestimmen kann.

$$t = \begin{pmatrix} (mouse.x - Endpoint.x * StartPoint.x - Endpoint.x) + \\ (mouse.y - Endpoint.y * StartPoint.y - Endpoint.y) + \\ (mouse.z - Endpoint.z * StartPoint.z - Endpoint.z) \end{pmatrix} \frac{1}{distance^2}$$

Abbildung: 96

$$d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

Abbildung: 97

Die Formel wird wie folgt umgesetzt

```
distance = Vector3.Distance(transform.position, EndPoint.position);

refObject = selecter.mousePosition;

//Get the intersection phase t
t = ((refObject.x - EndPoint.position.x) * (transform.position.x - EndPoint.position.x) +
     (refObject.y - EndPoint.position.y) * (transform.position.y - EndPoint.position.y) +
     (refObject.z - EndPoint.position.z) * (transform.position.z - EndPoint.position.z)) / (distance * distance);

t = Mathf.Clamp(t, 0f, 1f);
```

Abbildung: 98

Ein Punkt auf der Linie mit z.B. t = 0.25 ist 25% der Gesamtstrecke vom Startpunkt und 75% vom Endpunkt entfernt. Mit dem Wert t kann dann mit dem Code in Abbildung 100 ein Punkt berechnet werden, der im rechten Winkel zur Mausposition und zur Geradengleichung liegt.

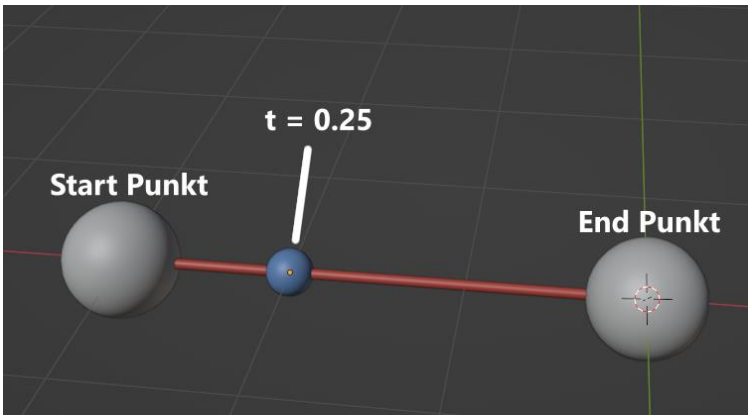


Abbildung: 99

Damit haben wir den Punkt der Geradengleichung „Schnittpunkt“, der der Mausposition am nächsten liegt. Mit der Funktion Vector3.Distance() kann man den projizierten Punkt der Maus und den Punkt auf der Geraden eingeben und erhält den Abstand zur Geraden. (Q30)

```
//set the intersection point in line with t
intersection.x = EndPoint.position.x + t * (transform.position.x - EndPoint.position.x);
intersection.y = EndPoint.position.y + t * (transform.position.y - EndPoint.position.y);
intersection.z = EndPoint.position.z + t * (transform.position.z - EndPoint.position.z);

//distance to green line
distanceToLine = Vector3.Distance(refObject, intersection);
```

Abbildung: 100

Die Linien zwischen den Punkten hängen von den anderen Punkten ab.

```
void regenerateLines()
{
    if (facePoints.Count > 2)
    {
        for (int i = 0; i < facePoints.Count - 1; i++)
        {
            DistanceToLine scriptLine;
            if (facePoints[i].TryGetComponent(out scriptLine))
            {
                scriptLine.EndPoint = facePoints[i + 1].transform;
            }
            else
            {
                facePoints[i].AddComponent<DistanceToLine>();
                facePoints[i].GetComponent<DistanceToLine>().EndPoint = facePoints[i + 1].transform;
            }
        }
        //add to last
        DistanceToLine scriptLineLast;
        if (facePoints[facePoints.Count - 1].TryGetComponent(out scriptLineLast))
        {
            scriptLineLast.EndPoint = facePoints[0].transform;
        }
        else
        {
            facePoints[facePoints.Count - 1].AddComponent<DistanceToLine>();
            facePoints[facePoints.Count - 1].GetComponent<DistanceToLine>().EndPoint = facePoints[0].transform;
        }
    }
}
```

Abbildung: 101

Daher müssen die Linien jedes Mal aktualisiert werden, wenn ein Punkt hinzugefügt oder entfernt wird.

Dies geschieht mit der Methode regenerateLines() in Abbildung 101. Diese beginnt mit einer Bedingung, die prüft, ob die Anzahl der Punkte in der Liste "facePoints" größer als 2 ist. Ist dies der Fall, wird eine Schleife über jedes Element von "facePoints" bis zum vorletzten Element gestartet (also "< facePoints.Count - 1").

Innerhalb der Schleife wird eine neue Variable "scriptLine" vom Typ "DistanceToLine" deklariert. Im nächsten Schritt wird überprüft, ob das aktuelle Element in der Liste "facePoints" das Skript "DistanceToLine" hat oder nicht. Wenn das Skript vorhanden ist, wird das EndPoint-Attribut des Skripts auf die Transformation des nächsten Elements in der Liste gesetzt. Wenn das Skript nicht vorhanden ist, wird ein neues Skript hinzugefügt und das EndPoint-Attribut auf das Transform des nächsten Elements in der Liste gesetzt.

Am Ende der Schleife wird das EndPoint-Attribut des letzten Elements in der Liste (facePoints[facePoints.Count-1]) auf das Transform des ersten Elements (facePoints[0]) gesetzt, um eine geschlossene Form zu erzeugen.

Erstellung eines 3D-Kniemodells:

Abbildung: 102

Für die Simulation einer Knieoperation wird ein anatomisch korrektes Kniemodell benötigt, an dem die Operation durchgeführt wird. Es gibt fertige medizinische Kniemodelle, die von medizinischem Personal zur Veranschaulichung oder zum Üben der Operation verwendet werden. Diese sind in der Regel sehr teuer. Da ich bereits fortgeschrittene Kenntnisse in der 3D-Modellierung besitze, habe ich mich entschieden, das Kniemodell selbst in Blender zu modellieren. Dieses kann dann später im richtigen Maßstab mit einem 3D-Drucker ausgedruckt werden. Da ich kein Arzt bin und auch nicht viel Ahnung von Anatomie habe, habe ich mich bei der Umsetzung von Dr. Bertrams unterstützen lassen. Anhand von Bildern habe ich erste Entwürfe entwickelt und das Kniemodell modelliert. Wenn ich mit einem Entwurf fertig war, schickte ich ihn an Dr. Bertrams, der mir Verbesserungen schrieb. So sind viele verschiedene Versionen des Kniemodells entstanden.

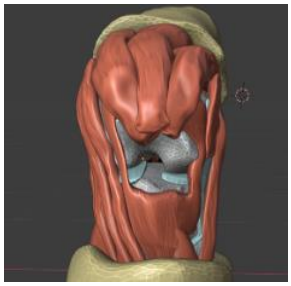


Abbildung: 103

Die erste Version beinhaltete das ganze Bein, den Knieknochen, die Muskelfasern und die Sehnen. Diese Version war noch weit davon entfernt, anatomisch korrekt zu sein. Die Muskelfasern waren nicht korrekt, die Sehnen falsch positioniert und die Kniescheibe konnte weggelassen werden, da dort die Operation stattfinden sollte. Um das Modell zu verbessern und anatomisch korrekter zu machen, habe ich die Muskelfasern noch einmal komplett überarbeitet. Außerdem ist es für die Operation besser, wenn das Knie angewinkelt ist, damit der Roboterarm besser greifen kann. Bei dieser Version habe ich mir die Muskelstrukturen genau angeschaut und diese übernommen.

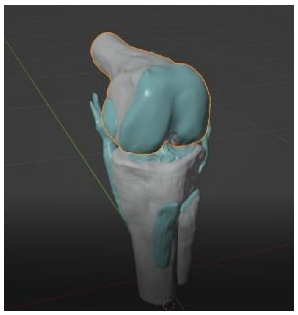


Abbildung: 104

Dr. Bertrams fand diese Version schon besser. Aber die ganzen Muskeln waren doch etwas überflüssig und verdeckten das Operationsfeld zu sehr.

Bei der nächsten Version habe ich noch einmal von vorne angefangen und mich ganz auf den Knieknochen konzentriert. Dabei habe ich mir die Sehnen genau angesehen und den Knorpel dazu modelliert. Das war schon viel besser. Jetzt ging es darum, die Anatomie des Knochens zu verbessern, die immer noch nicht ganz stimmte. Zum Beispiel war die vordere Knorpelfläche zu groß und überhaupt ist der Knieknochen zu groß.

Abbildung: 105

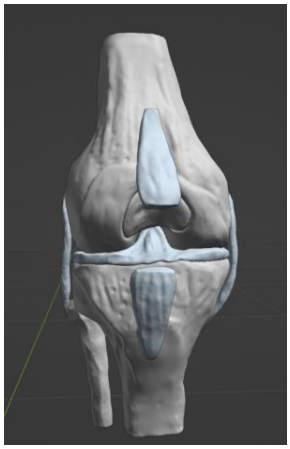


Abbildung: 106

Bis jetzt habe ich Online-Bildreferenzen verwendet, um das Knie zu modellieren. Bei der 3D-Modellierung kann man mit Bildreferenzen die meisten Details und die allgemeinen Proportionen gut erfassen und nachmodellieren. Da man nur bestimmte Perspektiven vom Modell bekommt, können die Proportionen etwas verloren gehen. Deshalb ist es am besten, das zu modellierende Objekt vor sich zu haben. Dafür hat mir der Chirurg ein Modell eines Knieknochens geliehen (Abbildung 105), damit ich es als Referenz

benutzen kann und die Proportionen viel besser sehe. So kann ich auch kleine Details besser erkennen und komme der richtigen Proportion sehr nahe. Laut Dr. Bertrams sieht das fertige Modell (Abbildung 106) genau wie das echte Knie aus. Ich habe das Kniemodell nicht schräg modelliert, weil ich die Operation von oben auf das Knie machen kann. Das Knie liegt also horizontal. Bei dem Kniemodell ist an einer Stelle vorne am Knorpel der geschädigte Bereich, den man herausnehmen und durch das Metallimplantat ersetzen kann.

Bei meinem Modell drucke ich auch diesen Einsetzer aus, der dann eingesetzt werden kann. Die beschädigte Stelle möchte ich mit Modelliermasse ausfüllen. Dazu modelliere ich eine Form (Abbildung 107), in die ich die Knete in Form des Einsetzers pressen kann. Der Roboterarm soll dann mit dem Bohrer den Knetteil entfernen und anschließend das Implantat an der Stelle einsetzen. So soll die Operation simuliert werden.

Abbildung: 107

Das Kniemodell ist wegen der verschiedenen Sehnen kompliziert zu drucken, deshalb teile ich es in Teile auf und drucke sie einzeln. Auf diese Weise kann ich auch verschiedene Filamente verwenden, um den Sehnen und der Einlage unterschiedliche Farben zu geben.

Die Teile können nun zusammengeklebt werden und das Modell ist vorerst fertig. Beim Bohren würde es sich aber sicherlich verschieben. Das wäre nicht gut, denn ich will das Modell ja auch in der Software virtuell synchron abgebildet haben und dann stimmt die Kalibrierung an sich nicht mehr.

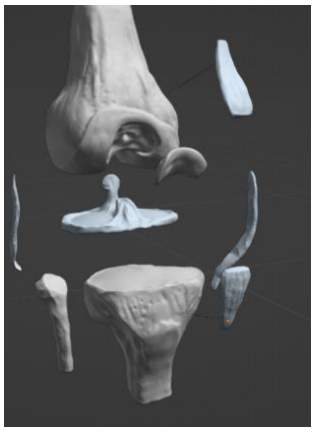


Abbildung: 108

Abbildung: 109

Deshalb entwerfe ich zusätzlich zum Kniemodell eine Plattform, in die das Knie eingelegt wird. Diese soll an der Seite eine Markierung haben, die den Startpunkt für die Kalibrierung festlegt.

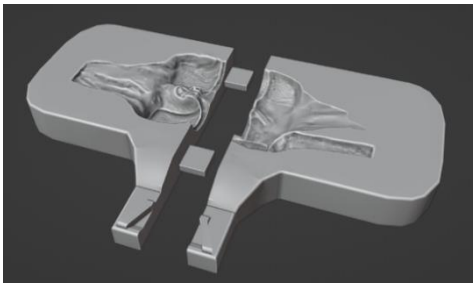


Abbildung: 110

Um einen perfekten Abdruck des Knies in der Plattform zu erhalten, kann ich den Abdruck in Blender mit dem booleschen Werkzeug erzeugen. Da die Platform größer als die Druckplatte ist, muss sie in zwei Teilen gedruckt werden, die dann zusammengeklebt werden.

Abbildung: 111

Das Knie hatte aber noch nicht den gezielten Effekt für das Projekt, deshalb habe ich das ganze Bein modelliert. Dazu habe ich mir eine Vorlage (Q31) aus dem Internet genommen und diese in Blender weiter weitere Details hinzugefügt. Mit einem Rig habe ich das Bein in eine Stellung gebracht, in der der die Operation durchgeführt werden kann.

Abbildung: 112

Abbildung: 113

Damit es gut steht, habe ich am Oberschenkel und am Fuß eine ebene Fläche hinzugefügt. Die Operation soll am Knieknochen durchgeführt werden, deshalb möchte ich dort die Haut und das Fleisch offen modellieren. Das kann man gut mit dem Sculpt-Werkzeug in Blender machen, mit dem man Geometrien wie Knetmasse verformen kann.

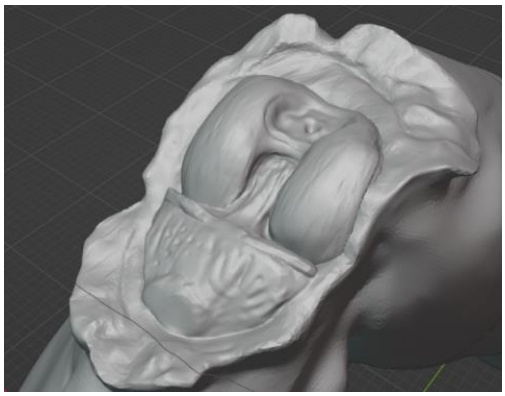


Abbildung: 114

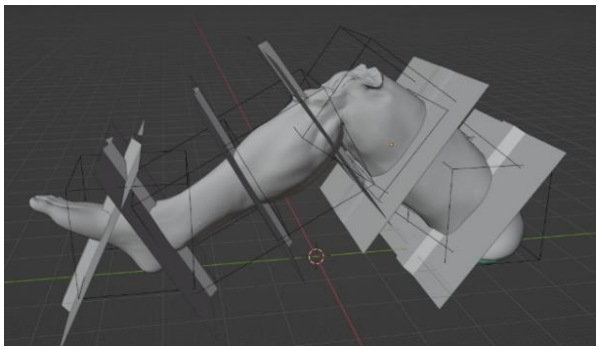


Abbildung: 115

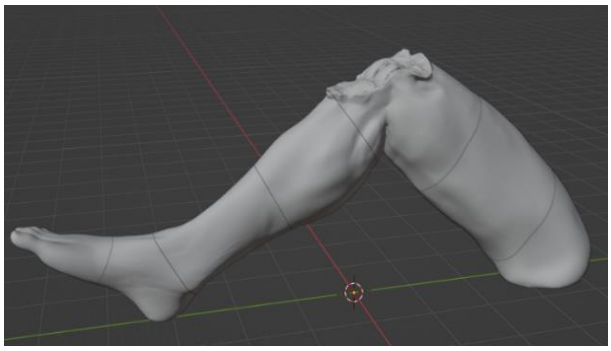


Abbildung: 116

Es war nicht möglich, es so zu drucken, also musste ich es in einzelne Teile aufteilen. Dazu habe ich die maximale Fläche des 3D-Druckers in einem Würfel genommen und angepasst, wie ich die Schnitte platzieren muss, damit die Schnitte im Druckbereich liegen (Abbildung 115).

Das Bein ist also in Einzelteile zerlegt, die ich jetzt drucken und später zusammenkleben kann. Um sie besser drucken zu können, lege ich sie gerade auf den Boden, damit die Druckrichtung mit der Vorlage übereinstimmt.

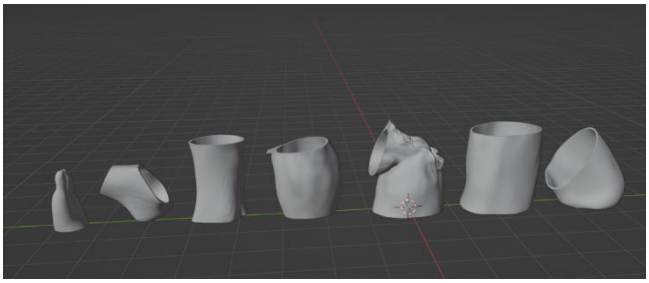


Abbildung: 117

Fazit:

Als ich die Projektarbeit "Entwicklung und Konstruktion eines chirurgischen Roboterarms zur Simulation einer roboterassistierten Patellofemoralgelenkersatzoperation mittels eines virtuellen Zwillings als Steuereinheit" durchführte, war ich von der Komplexität und dem Innovationspotenzial des Projekts begeistert.

Der Entwicklungsprozess war herausfordernd, aber letztendlich sehr lohnend. Ich musste mich mit verschiedenen Disziplinen wie Mechanik, Elektronik, Softwareentwicklung und Medizin auseinandersetzen, um eine effektive Lösung zu entwickeln.

Wie komplex zunächst einfach erscheinende Sachverhalte werden können, wenn man sich intensiv damit beschäftigt, habe ich während des Entwicklungsprozesses schnell gemerkt. Gerade wenn man lange an einem Thema arbeitet, muss man den Überblick behalten und konzentriert an Lösungen arbeiten. An dem Projekt hat mich besonders der Aspekt interessiert, dass man mit Software physikalische Vorgänge durch Motoren beeinflussen kann. Das hat mich unter anderem dazu motiviert, mich in die technischen Details der Motoren, der Mikrochips und der Stromversorgung einzuarbeiten.

Faszinierend fand ich auch die Mechanik des Roboterarms. Am Anfang war es sehr faszinierend, diese ungewöhnliche Mechanik des Cycloidal Drive-Prinzips zu erlernen. Es war eine Herausforderung, die verschiedenen Komponenten zu verstehen und zu visualisieren, wie sie zusammenwirken, um die gewünschte Bewegung zu erzeugen. Es war eine gute Gelegenheit, meine Analyse- und Problemlösungsfähigkeiten zu verbessern. Am Ende war es für mich sehr befriedigend, nicht nur verstanden zu haben, wie der Cycloidal Drive funktioniert, sondern ihn auch in meinem Projekt einsetzen zu können und zu sehen, wie er nach vielen Fehlversuchen funktioniert.

Außerdem habe ich nicht nur viel über Elektronik und die Programmierung von Mikrochips gelernt, sondern auch viel über Softwareentwicklung. In diesem Projekt habe ich intensiv an Lösungen für Probleme gearbeitet, die ich vor einem Jahr noch für unlösbar gehalten hätte, die ich aber lösen konnte. Dadurch wurde mir noch bewusster, wie man seinen Code plant und strukturiert. Gerade bei einem so großen Projekt wie diesem habe ich gemerkt, dass wenn man am Anfang nicht auf Ordnung achtet, es sehr schnell unübersichtlich wird.

Während der Softwareentwicklung habe ich viele wichtige Fähigkeiten erworben, wie beispielsweise das Schreiben von effizientem Code und das Debuggen von Fehlern. Ich habe auch gelernt, wie man komplexe Algorithmen entwirft und implementiert, um komplexe Aufgaben zu lösen. Dies war eine unglaublich wertvolle Erfahrung, da ich jetzt in der Lage bin, meine Programmierfähigkeiten auf eine breitere Palette von Projekten anzuwenden.

Ich hatte auch die Möglichkeit, verschiedene Entwicklungsplattformen und Tools kennenzulernen, wie z.B. Arduino IDE, Visual Studio und Unity. Diese Erfahrung hat mir geholfen, meine Fähigkeiten zu erweitern und mich auf die neuesten Entwicklungen in der Softwareentwicklung zu konzentrieren.

Abbildung: 118

Bei dem anatomischen Kniemodell haben mich Prof. Dr. Sebastian Gehrmann und Dr. Bertrams sehr unterstützt und ohne die finanzielle Hilfe des Fördervereins des Gymnasiums in den Filder Benden und der Katholischen Karl-Leisner-Trägergesellschaft in Kleve wäre das Projekt nicht möglich gewesen. Danken möchte ich Herrn Prof. Dr. Sebastian Gehrmann und Herrn Dr. Bertrams, meinem Mentor Herrn Lachmann für die gemeinsame Ideenfindung zu diesem Projekt sowie dem Förderverein des Gymnasiums in den Filder Benden und der Katholischen Karl-Leisner-Trägergesellschaft.

Insgesamt war die Projektarbeit eine prägende Erfahrung, die mir nicht nur fachliche Kompetenzen vermittelt hat, sondern auch, wie wichtig Projektmanagement ist und dass man sich Hilfe holen muss, wenn man wirklich nicht weiterkommt. Es hat Spaß gemacht, an einem Projekt zu arbeiten, das so viele Bereiche vereint. Am Ende ein funktionierendes Produkt zu haben, ist ein Gefühl, auf das man stolz sein kann.

Katholische
Karl-Leisner-Trägergesellschaft

Abbildung: 119

gfb
GEMEINSAM
FÜR BILDUNG

Förderverein des Gymnasiums in den Filder Benden e. V.
WIR UNTERSTÜTZEN DIE SCHÜLERSCHAFT UND DIE SCHULE

Abbildung: 120

Quellenverzeichnis:

Agile Projektentwicklung:

- Q1: https://berufsinformatik.de/agile-methoden-und-projekte/

Arduino:

- Q2: https://www.arduino.cc/en/Guide/Introduction
- Q3: https://thecustomizewindows.com/2018/05/difference-between-analog-and-digital-pins-in-arduino-uno/

PWM:

- Q4: https://www.kompulsa.com/introduction-pwm-pulse-width-modulation-works/

Arduino IDE:

- Q5: https://docs.arduino.cc/learn/starting-guide/the-arduino-software-ide
- Q6: https://www.arduino.cc/reference/en/libraries/servo/
- Q7: https://docs.arduino.cc/learn/electronics/servo-motors

Unity:

- Q8: https://de.wikipedia.org/wiki/Unity_(Spiel-Engine)
- Q9: https://assetstore.unity.com/packages/tools/integration/ardity-arduino-unity-communication-made-easy-123819

PCA-Chip:

- Q10.1: https://cdn-shop.adafruit.com/datasheets/PCA9685.pdf
- Q10.2: https://learn.adafruit.com/16-channel-pwm-servo-driver/using-the-adafruit-library

Blender:

- Q11: https://de.wikipedia.org/wiki/Blender_(Software)

Rigging:

- Q12: https://conceptartempire.com/what-is-rigging/

Onshape:

- Q13: https://de.wikipedia.org/wiki/Onshape

Motoren:

- Q14: https://www.leifiphysik.de/elektrizitaetslehre/kraft-auf-stromleiter-e-motor/grundwissen/elektromotor
- Q15: https://howtomechatronics.com/tutorials/arduino/stepper-motors-and-arduino-the-ultimate-guide/
- Q15.2: https://de.wikipedia.org/wiki/Schrittmotor#Baugr%C3%B6%C3%9Fe
- Q16: https://howtomechatronics.com/how-it-works/how-servo-motors-work-how-to-control-servos-using-arduino/

EEZYbotARM MK2:

- Q17: http://www.eezyrobots.it/eba_mk2.html

Inverse Kinematik und Algorithmen:

- Q18.1: https://ww2.mathworks.cn/discovery/inverse-kinematics.html
- Q18.2: https://www.youtube.com/watch?v=3s7h2MHQtxc&t=317s

RoTechnic: Cycloidal Drive

- Q19: https://www.youtube.com/@roTechnic

Cycloidal Drive:

- Q20: https://www.tec-science.com/mechanical-power-transmission/planetary-gear/how-does-a-cycloidal-gear-drive-work/
- Q21: https://www.youtube.com/watch?v=r2TWC7vTdvs&t=84s

Elektronik:

- Q22.1: https://www.utmel.com/components/how-to-interface-tmc2209-stepper-driver-with-microcontroller?id=903
- Q22.2: https://all3dp.com/2/arduino-cnc-shield/
- Q22.3: https://www.circuitspecialists.com/blog/stepper-motor-power-supplies/

PE-Leiter:

- Q23: https://www.helpster.de/l-stromkabel-bedeutung_195377

H-Brücke:

- Q24: https://www.e-hack.de/l298-dualer-vollbruckentreiber-funktion-schaltung/
- Q25: https://www.build-electronic-circuits.com/h-bridge/

AccelStepper Bibliothek:

- Q26: http://www.airspayce.com/mikem/arduino/AccelStepper/classAccelStepper.html

FastIK Add on:

- Q27: https://assetstore.unity.com/packages/tools/animation/fast-ik-139972

Waypoint System:

- Q28: https://www.youtube.com/watch?v=EwHiMQ3jdHw&t=1641s

Flächeneditor:s

- Q29: https://www.youtube.com/watch?v=bPO7_JNWNmI

Strecke von Punkt zu Linie Skript:

- Q30: https://www.youtube.com/watch?v=CiAwRA6IQi0

Bein Modell Vorlage:

- Q31: https://free3d.com/3d-model/leg-01-v1--546521.html

Bildquellenverzeichnis:

Eigene Fotografien:

12, 16, 26, 27, 36, 42, 61, 105, 107, 109, 111, 118

Eigene Screenshots:

6, 8, 10, 11, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 28, 32, 33, 34, 35, 37, 38, 41, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 59, 65, 66, 67, 68, 69, 70, 71, 72, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 98, 99, 100, 101, 102, 103, 104, 106, 108, 110, 112, 113, 114, 115, 116, 117

Abbildung 1: https://www.drklotz.org/roboter-unterstuetzte-knieprothese

Abbildung 2: https://www.yeovilorthoclinic.co.uk/knee-arthritis-and-stem-cell-therapies/

Abbildung 3: fotografiert von Marc Lachmann auf der Medica Messe

Abbildung 4: https://www.teachmemicro.com/wp-content/uploads/2017/03/arduino-pins.png

Abbildung 5: https://www.researchgate.net/figure/PWM-signal-with-its-two-basic-time-periods_fig4_271437313

Abbildung 7: https://www.maxphi.com/wp-content/uploads/2017/08/led-interfacing-arduino-wiring-768x378.png

Abbildung 9: https://i.pinimg.com/736x/4a/6e/3f/4a6e3f3a0e5ff69d1b0e4a0e50d1fafe.jpg

Abbildung 29: https://imgr1.auto-motor-und-sport.de/Elektromotor--169Gallery-bd3e6b39-1801763.jpg

Abbildung 30: https://howtomechatronics.com/wp-content/uploads/2022/05/Stepper-Motor-Working-Principle-1536x484.png?ezimgfmt=ng:webp/ngcb2

Abbildung 31: https://howtomechatronics.com/how-it-works/how-servo-motors-work-how-to-control-servos-using-arduino/

Abbildung 39: https://ww2.mathworks.cn/discovery/inverse-kinematics/_jcr_content/mainParsys/image.adapt.full.medium.jpg/1668067816271.jpg

Abbildung 40: https://i.stack.imgur.com/9JNTu.jpg

Abbildung 43: https://youtu.be/0hGy4AxUOnk?t=258

Abbildung 44: https://youtu.be/r2TWC7vTdvs?t=276

Abbildung 45: https://i.stack.imgur.com/eWCem.gif

Abbildung 57 https://temperosystems.com.au/wp-content/uploads/2021/09/TMC2209-Stepper-Motor-Driver.jpg

Abbildung 58: https://starthardware.org/wp-content/uploads/2020/12/Arduino-drv8825-stepper.gif

Abbildung 60: https://www.roboter-bausatz.de/media/image/81/bb/7b/3D241-4.jpg

Abbildung 62: https://www.e-hack.de/wp-content/uploads/2021/10/l298_modul_pinbelegung.jpg

Abbildung 63,64: https://www.build-electronic-circuits.com/h-bridge/

Abbildung 73: https://i.stack.imgur.com/lKN6o.jpg

Abbildung 97: https://www.chilimath.com/lessons/advanced-algebra/distance-between-point-and-line-formula/

Abbildung 119: https://static.kkle.de/assets/Logos/kkle-traeger.svg

Abbildung 120: https://gfb-der-förderverein.de/wer-wir-sind

Alle Internetquellen im Quellen-und Bildverzeichnis wurden zuletzt am 29.03.2023 überprüft.